VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

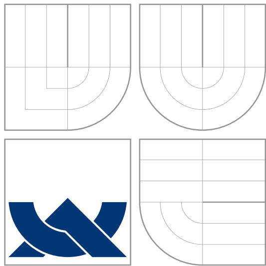
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KONTROLA PROGRAMU ZALOŽENÁ NA SLEDOVÁNÍ SYSTÉMOVÝCH VOLÁNÍ

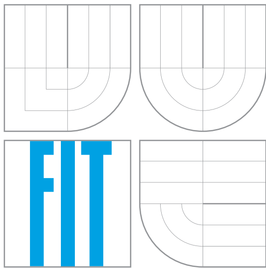
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE TOMÁŠ PRVÁK
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KONTROLA PROGRAMU ZALOŽENÁ NA SLEDOVÁNÍ SYSTÉMOVÝCH VOLÁNÍ

PROGRAM TESTING VIA SYSTEM CALL MONITORING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ PRVÁK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2010

Abstrakt
Táto práca sa zaoberá problematikou analýzy systémových volaní v operačnom systéme Linux a jej použitím na overenie validity programov. Dozvieme sa o princípe, na akom fungujú systémové volania, a taktiež bude spomenutý nástroj strace, pomocou ktorého je možné tieto volania sledovať a bližšie sa pozrieme ako jednotlivé linuxové grafické prostredia využívajú systémové volania. Bude navrhnutý a implementovaný nástroj pre overenie validity programov. Na záver sa oboznámime so spôsobom, ako vytvoriť pre tento nástroj pravidlá, ktoré rozšíria jeho možnosti.

Abstract
This work deals with the analysis of system calls in Linux and how they can be used to verify the validity of the programs. We learn about the principle how system calls works, we will discuss the strace tool, through which one can monitor these calls, and we also make a closer look at how the different Linux graphical environments use system calls. A design and the implementation of o tool for checking the corretness of a program will be presented.The work also discuss the way how to extend the tool with the user defined correctness rules.

Klíčová slova
softwarové testovanie, opakovaná analýza, grey-box testovanie, systémové volania.

Keywords
software testing, repeated analysis, grey-box testing, system calls.

Citace
Tomáš Prvák: Kontrola programu založená na sledování systémových volání, bakalářská práce, Brno, FIT VUT v Brně, 2010

Kontrola programu založená na sledování systémových volání

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Aleša Smrčku. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

....
Tomáš Prvák
14. mája 2010

Poděkování

Moje poďakovanie patrí vedúcemu tejto práce Ing. Alešovi Smrčkovi, ktorý mi pri konzultáciách počas celého semestra vždy poskytol cenné rady a pripomienky.

© Tomáš Prvák, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod 3

2 Základy testovania 4
2.1 Analýza programu 4
2.1.1 Statická analýza 4
2.1.2 Dynamická analýza 5
2.2 Testovacie prístupy 5
2.2.1 Metóda black-box 5
2.2.2 Metóda white-box 7
2.2.3 Metóda grey-box 7

3 Systémové volania a ich sledovanie 9
3.1 Systémové volania 9
3.2 Strace 10
3.2.1 Použitie v praxi 10
3.2.2 Ukážkový výstup 11
3.3 Ďalšie programy 12
3.3.1 Dtrace 12
3.3.2 Truss 12
3.3.3 Ktrace 13
3.3.4 SystemTap 13
3.4 Experiment 13
3.4.1 Testovací scenár experimentu č.1 13
3.4.2 Výsledok experimentu č.1 14
3.4.3 Testovací scenár experimentu č.2 16
3.4.4 Výsledok experimentu č.2 16

4 Návrh programu 17
4.1 Špecifikácia požiadaviek 17
4.2 Návrh testovacieho systému 18

5 Implementácia programu 20
5.1 Použité nástroje 20
5.2 Program 21
5.2.1 Analyzovanie a parsing výstupu strace 21
5.2.2 Vyhodnotenie výstupu strace 22
5.2.3 Interaktívny mód 23

6 Pravidlá a jazyk **24**
6.1 Ukážka pravidla 24
6.2 Vytvorenie všeobecného pravidla 25

7 Záver **27**

A Príklad testu **29**

B Obsah CD **32**

Kapitola 1

Úvod

Táto práca sa zaoberá problematikou a spracovaním systémových volaní v operačnom systéme Linux a ich analýzou. Testovanie tvorí významnú súčasť každého procesu vývoja softvéru. Podľa G. J. Myersa [12] predstavuje testovanie približne polovicu celkového času a viac ako polovicu celkových nákladov potrebných na vývoj systému. Tieto vysoké čísla nasvedčujú tomu, že testovanie je naozaj dôležité. Napriek tomu sa často prehliada, nevenuje sa mu dostatočná pozornosť alebo sa kvôli časovému sklzu projektu nevykonáva tak dôsledne, ako by bolo potrebné. Treba si pritom uvedomiť, že testovanie ovplyvňuje výslednú kvalitu výrobku. Jeho cieľom je odhaliť prípadné chyby ešte skôr ako sa výrobok dostane k používateľovi alebo spôsobí škody.

Úlohou tejto práce je navrhnúť a implementovať program, ktorý by dokázal otestovať iný program a zistiť, či počas jeho behu nedošlo k chybám na úrovni systémových volaní.

Úvodná časť textu sa zaoberá pojmom testovanie, zameriame sa na problematiku black-box, white-box a grey-box testovania, vysvetlíme si, ako pracujú systémové volania v Linuxe a pozrieme sa na programy, ktoré dokážu analyzovať programy na úrovni systémových volaní. Taktiež bude analyzovaný experiment, v ktorom sa pokúsim porovnať využitie systémových volaní v rôznych Linuxových grafických rozhraniach. Ústrednými kapitolami práce sú časti, v ktorých sa nachádza návrh programu, jazyka a interpreta pre zadávanie jednoduchých otázok nad postupnosťou systémových volaní a jeho implementácia. Na záver sa pokúsim vysvetliť spôsob, akým bude možné pre tento program vytvoriť testovacie pravidlá.

Kapitola 2

Základy testovania

Táto kapitola objasňuje niektoré základné pojmy z oblasti testovania software. Vysvetlíme si rozdiely medzi statickou a dynamickou analýzou programu a bližšie sa pozrieme na testovacie prístupy ktoré sa používajú pri testovaní.

2.1 Analýza programu

Analýza počítačového programu je proces automatického analyzovania chovania programu. Analýza programu sa delí na dve hlavné kategórie, ktorými sú statická a dynamická analýza. Hlavnou úlohou tejto analýzy je dosiahnutie optimalizácie a korektnosti počítačového programu. Informácie v tejto podkapitole sú voľne prevzaté z [13].

2.1.1 Statická analýza

Jej cieľom je kontrola kódu so zameraním na jeho správnosť a prehľadnosť. Kontroluje sa tok dát v programe, aby sa vylúčili prípadné chyby v zachádzaní s dátami a pamäťou. Postupuje sa nasledovným spôsobom: najprv sa urobí bežná kontrola kódu, overenie jeho úplnosti, kontrola prekladu s pridaním informácií pre odlaďovací systém (debugger) a kontrola všetkých varovaní kompilátora. V druhej fáze sa pristúpi ku kontrole prehľadnosti zdrojového kódu a jeho vhodného delenia do funkcií a modulov. Potom nasleduje vyhľadanie toku dát v programe a jeho oddelenie od ostatných častí programu. Urobí sa kontrola všetkých vetiev a cyklov súvisiacich s tokom dát a kontrola zabezpečenia dát. Ďalšou etapou statickej analýzy je kontrola modularity programu, teda toho, ako sú v ňom oddelené funkcie zaoberajúce sa rôznymi formátmi spracovania dát a ako sú spracované do samostatných modulov. Pritom je potrebné skontrolovať, či dochádza k interakcii jednotlivých modulov len očakávaným spôsobom. Potom sa pristúpi ku kontrole modulov, ktorá spočíva v oddelení samostatných častí kódu zaoberajúcich sa spracovaním dát a statickom simulovaní vstupov a výstupov pre tieto časti programu. Nakoniec sa urobí kontrola ošetrenia výnimiek a neštandardných situácií programu.

Pre účely statickej analýzy je vhodné použiť softvérové nástroje pre sledovanie toku dát v programe, odlaďovacie programy (debugger) a programy kontrolujúce prácu s pamäťou.

2.1.2 Dynamická analýza

Cieľom dynamickej analýzy je overiť funkciu a stability programu a jeho zhodu s dokumentáciou pomocou black-box, white-box alebo grey-box testovania. Test by mal byť urobený priamo v prostredí, v ktorom sa bude program používať, poprípade v podobnom simulovanom prostredí. Pri dynamickej analýze sa postupuje nasledujúcim spôsobom:

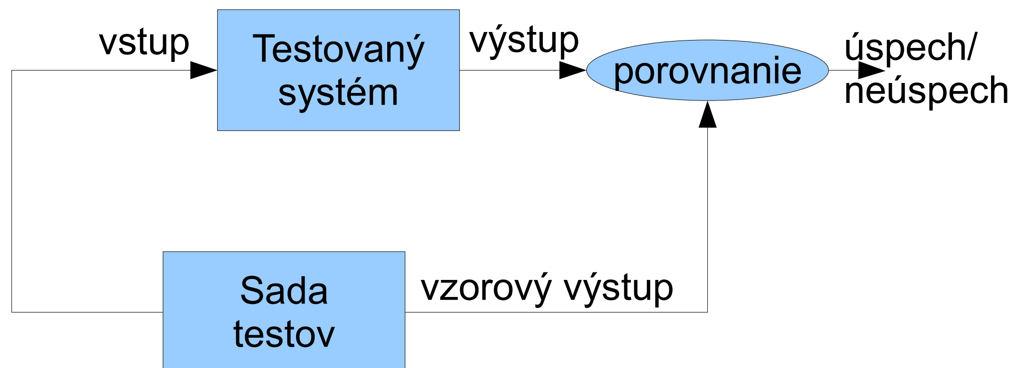
- Identifikuje sa binárna verzia programu napríklad pomocou hašovacej funkcie a zálohuje sa kópia programu ako referencia aktuálnej validovanej verzie
- Je predvedené typické užitie programu s ohľadom na bežne vykonávané operácie. V prípade, že ide o program, ktorý je určený k ďalšej distribúcií, je testovaná taktiež inštalácia programu a jeho inicializácia.
- V prípade white-box testu sa skontrolujú všetky vetvy a cykly programu a to aspoň jedným priechodom
- Vyvolajú sa kritické situácie, dané napríklad chýbajúcim hardware či chybnými vstupnými dátami a skontroluje sa, ako sú ovplyvnené vstupné dáta, ako program zaznamenáva kritické situácie a či je možné vzniknuté záznamy modifikovať
- Nakoniec sa vykoná kontrola dokumentácie programu a porovnanie zo skutočným stavom.

2.2 Testovacie prístupy

Testovanie, ktorým sa budem v tejto práci zaoberať, bude na dáta nahliadať ako na grey-box testovanie. Aby sme však pochopili, čo to grey-box testovanie je, musíme vysvetliť testovacie prístupy, z ktorých sa grey-box testy skladajú. Informácie v tejto kapitole sú voľne prevzaté z [5] a [9].

2.2.1 Metóda black-box

Black box testovanie, v anličtine taktiež známe ako opaque box, closed box, behavorial box alebo funkčné testovanie je realizované bez znalosti vnútornej dátovej a programovej štruktúry.



Obrázok 2.1: Black-box testovanie z pohľadu testera.

To znamená, že tester nemá k dispozícii zdrojové kódy, disponuje iba binárnym kódom a špecifikáciou rozhrania. Tento spôsob testovania vyžaduje testovacie scenáre, ktoré sú buď poskytnuté testerovi alebo si ich tester u niektorých typov testov sám vytvorí. Vzhľadom na to, že sú obvykle definované typy a rozsahy hodnôt prípustných a neprípustných pre danú aplikáciu a tester vie, aký zadal vstup, tak vie aj aký výstup alebo chovanie môže od aplikácie očakávať. Black box testy môžu prebiehať ručne, alebo automatizovane za použitia najrôznejších nástrojov. Aj v tomto prípade sa z obľubou využívajú obidva prístupy. Black box sa javí ako ideálny tam, kde sú presne definované vstupy a rozsahy možných hodnôt.

Black box testy je možné využiť napríklad na zistenie problémov typu odopretia služby (DoS), alebo aktuálnych zraniteľností už bežiaceho systému alebo aplikácie. V prípade použitia tohto spôsobu testovania treba vždy počítať s určitým rizikom pádu daného systému. Black box testom je tiež možné demonštrovať chybu, ktorá bola objavená počas white box testu. K vykonaniu testov stačí mať k dispozícii len daný program alebo poznať jeho umiestnenie, v takomto prípade môže byť test vykonaný aj vzdialene po sieti

Výhody

- Jednoduchosť – test môže byť vykonaný bez znalosti programovacích jazykov.
- Rýchlosť – je možné rýchlo a v krátkom období otestovať rozsiahle systémy.
- Transparentnosť – test je pre zákazníka zrozumiteľný – chápe, čo a ako sa bude testovať, môže a často to býva i on, kto testovacie scenáre vytvára a testovanie potom sám vykonáva.
- Testovacie scenáre môžu byť napísané v okamihu, kedy je kompletná špecifikácia.
- Testovanie nie je založené na aktuálnej implementácii, aj keď sa zmení programovací jazyk, OS a HW, testovanie bude prebiehať stále rovnako. Testovací scenár nie je nutné meniť.
- Testerovi nie je nutné sprístupňovať zdrojový kód.

Nevýhody

- Nižšia kvalita testu – to, že sa na výstupe objaví očakávaná hodnota, neznamená, že aplikácia je napísaná správne. Kód može byť značne neefektívny.
- Nežiadúce chovanie testu – okrem požadovanej funkcionality môže produkt vykonávať aj iné akcie, ktoré nie sú v špecifikácii, ich prejav sa na výstupe neobjaví a test ich preto neodhalí.

2.2.2 Metóda white-box

White box testovanie, v angličtine známe tiež ako glass box, clear box, open box alebo tiež štrukturálne testovanie, predpokladá znalosť vnútornej štruktúry.

Lepšie povedané vyžaduje znalosť vnútorných dátových a programových štruktúr a tiež toho, ako je systém implementovaný. Testerovi sú v prípade white box testovania poskytnuté všetky informácie, to znamená, že má k dispozícii nielen príslušnú dokumentáciu, ale aj binárny a zdrojový kód testovanej aplikácie. Tester musí zdrojovému kódu porozumieť a analyzovať ho. Niekedy sa tomuto spôsobu testovania hovorí tiež audit zdrojového kódu. Zahraničná literatúra má pre túto činnosť označenie 'code-review exercise'. White box testovanie môže prebiehať automatizovane alebo ručne. V praxi sa však veľmi často tieto spôsoby vhodne kombinujú.

White box testovanie sa prevažne zameriava na tzv. pokrytie (angl. coverage). Napr. pokrytie kódu, pokrytie podmienok apod., pričom hlavnou myšlienkou je, že pokiaľ nejaký test pokryl (otestoval) daný kus kódu, potom je pravdepodobne kód správny.

Výhody

- Včasné odhaľovanie chýb – analýza zdrojového kódu umožní odhaliť chyby, ktorých sa programátor dopustil ešte skôr, ako bol kód skompilovaný.
- Odhalenie nežiadúceho kódu – treba si uvedomiť, že program môže okrem požadovaných operácií uskutočňovať i niektoré ďalšie nežiadúce operácie, ktoré môžu zostať počas iných testov nepovšimnuté.

Nevýhody

- Náročnosť – vyžaduje výbornú znalosť cieľového systému, testovacích nástrojov a programovacích jazykov.
- Vysoké náklady – požaduje špecializované nástroje ako sú analyzátory zdrojového kódu, debuggery atď.

2.2.3 Metóda grey-box

Grey box testovanie, známe taktiež ako translucent box predpokladá obmedzenú znalosť interných dátových a programových štruktúr za účelom navrhnutia vhodných testovacích scenárov, ktoré sa realizujú na úrovni black box.

Spôsob testovania je tak kombináciou black box a white box testovania. Nejedná sa o black box, pretože tester pozná niektoré vnútorné štruktúry, ale zároveň sa nejedná ani o white box, pretože znalosti vnútorných štruktúr nejdú do hĺbky. Koncept grey box testovania je veľmi jednoduchý. Ak tester vie, ako produkt funguje zvnútra, potom ho môže lepšie otestovať zvonku. Grey box test, rovnako ako black box je prevádzaný zvonku, ale tester je lepšie informovaný, ako jednotlivé komponenty fungujú a spolupracujú.

Výhody

- Zlučuje v sebe výhody black box i white box prístupu.
- Neintrusívny – prístup ku zdrojovému ani binárnemu kódu nie je potrebný. Je založený na znalosti funkčnej špecifikácie, rozhrania a architektúre aplikácie.
- Inteligentné testy – tester je schopný vďaka znalostiam, aj keď len obmedzeným, napísať inteligentné testovacie scenáre zamerané i na manipuláciu s dátami a použité komunikačné protokoly.

Nevýhody

- Neúplné otestovanie – binárne a ani zdrojové kódy nie sú k dispozícii a nie je tak možné otestovať všetky dátové toky. Miera pokrytia týchto tokov závisí na schopnostiach, znalostiach a skúsenostiach testera.
- Kvalita kódu – rovnako ako u black box testu, to že niečo funguje podľa špecifikácie a je to odolné proti známym zraniteľnostiam, neznamená, že kód je efektívny a že aplikácia neobsahuje žiaden nežiaduci kód.

Kapitola 3

Systémové volania a ich sledovanie

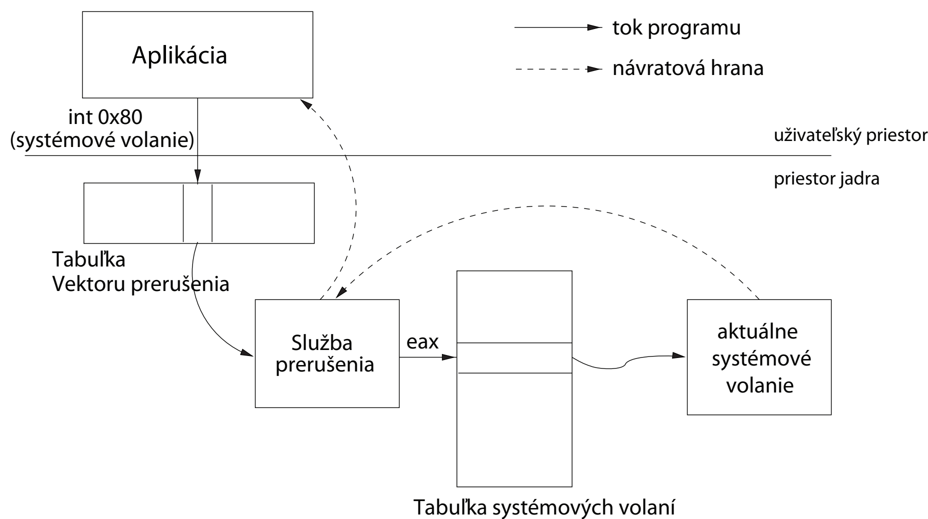
V tejto kapitole si priblížime pojem systémové volanie a pokúsim sa spraviť prehľad nástrojov, ktoré umožňujú sledovať systémové volania. Bližšie sa pozrieme predovšetkým na program `strace`, s ktorým sa budeme stretávať aj v ďalších kapitolách tejto práce a vysvetlíme si princíp jeho fungovania.

3.1 Systémové volania

Systémové volania (angl. system call, syscall) je mechanizmus používaný aplikáciami k volaniu funkcií operačného systému u jadier monolitického typu. Nájdeme ich u všetkých unixových systémoch. Systémy Microsoft Windows majú jadro typu mikrokernel a používajú výhradne medzi-procesorovú komunikáciu prostredníctvom Windows API, kde programátor nerozlišuje medzi knižničnou funkciou a využívaním služieb operačného systému.

V počítačovej terminológií by sa pojem systémové volania [10] dal vysvetliť ako žiadosť programu o službu z jadra operačného systému, ktoré zvyčajne nemá povolenie na spustenie. Systémové volania poskytujú rozhranie medzi procesom a operačný systémom. Väčšina operácií interaguje s požiadavkami do systému, ktoré nie sú dostupné na užívateľskej úrovni. Akékoľvek formy komunikácie s inými procesmi vyžadujú použitie systémových volaní.

Systémové volania sa generujú prostredníctvom prerušenia (napr. int 0x80 na Linux/i86). Tieto prerušenia predajú riadenie jadru, služba pre toto prerušenie použije tabuľku systémových volaní (nastavenú operačným systémom), ktorá je prehľadaná (použitím hodnoty z registra `eax` ako index) a riadenie je presunuté na kód odpovedajúci hľadanému systémovému volaniu v priestore jadra. Tento proces znázorňuje obrázok 3.1. Procesy môžu volať obsluhu jadra aj inými spôsobmi, ale pre potreby bežného programovania slúžia systémové volania. K zobrazeniu súpisu uskutočnených systémových volaní slúži spustenie programu `strace`, o ktorom sa dozvieme v nasledujúcej časti tejto kapitoly.



Obrázok 3.1: Architektúra systémových volaní v Linuxe (prevzaté z [14]).

3.2 Strace

Obvyklou metódou ladenia, je takzvané trasovanie, inak nazvané aj sledovanie priebehu programu. Trasovanie môže prebiehať na rôznych úrovniach, od riadkov zdrojového kódu až po volania jednotlivých inštrukcií procesoru. `Strace` pracuje na úrovni systémových volaní, takže s jeho pomocou je možné zistiť, čo program požadoval po jadre. Jednotlivé zaznamenané operácie môžu byť pokusy o otvorenie súboru, čítanie či zápis do neho alebo napríklad vytvorenie potomka. O každom volaní sa dozvieme, s akými argumentami bolo volané a taktiež aj to, ako dopadlo. To už je dosť na to, aby sa program `strace` stal veľmi šikovným pomocníkom pre rôzne problémové situácie.

Hlavnou výhodou použitia `strace` [6] je, že pre ladenie nemusíme mať k dispozícii zdrojové kódy, program nemusí byť kompilovaný a môže byť staticky i dynamicky linkovaný. `Strace` je možné bez problémov použiť napríklad v prípadoch, kedy máme k dispozícii len binárne súbory programu, ktorý nepracuje správne. Použitie `strace` je taktiež pre neprogramátora jednoduchšie ako použitie debuggeru.

3.2.1 Použitie v praxi

`Strace` je jednoduchý program, pri jeho používaní nie je potrebné poznať desiatky prepínačov a volieb. Úplne najjednoduchším príkladom použitia je táto syntax:

```
strace ls
```

V tomto prípade nástroj `strace` spustí program `ls` a zoznam systémových volaní vypíše na štandardný chybový výstup. Pokiaľ program sám niečo vypisuje na konzolu alebo chcete súpis volaní študovať neskôr, je vhodné presmerovať výstup `strace` do súboru. To môžete urobiť buď pomocou presmerovania v shelli (príklad pre bash):

```
strace ls 2>ls.strace
```

alebo môžete použiť priamo prepínač –o

```
strace -o ls.strace ls
```

v obidvoch prípadoch je výsledkom vytvorenie súboru ls.strace, ktorý po ukončení príkazu obsahuje zoznam systémových volaní

Ďalším, pre moju prácu zaujímavým prepínačom je prepínač -f. Slúži k zapnutiu sledovania potomkov hlavného procesu. Výsledný výpis sledovania tak bude obsahovať nielen systémové volania hlavného procesu, ale aj všetky systémové volania procesov, ktoré hlavný proces vytvoril.

3.2.2 Ukážkový výstup

Rád by som ešte objasnil, niektoré pravidlá systémových volaní, bez ktorých by ste sa pri ladení alebo testovaní nezaobišli. Návratové hodnoty volaní môžu byť rôzne, ale obvykle platí, že chyba je signalizovaná hodnotou -1 a úspech je signalizovaný nulovou hodnotou alebo kladným číslom. K presnejšiemu určeniu chyby slúži premenná `errno`, ktorú `strace` do výpisu taktiež zahrnie, a to dokonca aj s textom chybového hlásenia, takže orientácia je pomerne jednoduchá. Ak sa čitateľ bude chcieť o niektorom systémovom volaní dozvedieť viac, napríklad čo ktorý argument znamená, môže si prezrieť jeho manuálové stránky.

Teraz, keď už vieme, ako sa `strace` používa, môžeme sa pozrieť na jeho výstup. Časť výstupu je znázornená v tabuľke 3.1. Táto tabuľka znázorňuje ako vyzerá výstup programu `strace` pri spustení programu `false`:

```
execve("/bin/false", ["false"], [/* 35 vars */]) = 0
brk(0) = 0x8049c98
open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=35620, ...}) = 0
old_mmap(NULL, 35620, PROT_READ, MAP_PRIVATE, 4, 0) = 0x40018000
close(4) = 0
open("/lib/libc.so.6", O_RDONLY) = 4
...
...
_exit(1) = ?
```

Tabuľka 3.1: Ukážka výstupu programu strace

Úplne na začiatku vidíme, že novo vzniknutý proces bude nahradený programom `false`. Volanie `execve` preberá 3 parametre: plnú cestu k programu, pole jeho argumentov a pole premenných v prostredí. Ďalej sa dozvedáme, že volanie dopadlo úspešne, pretože práve toto volanie vracia pri úspechu nulovú návratovú hodnotu. Po volaní `execve` nasleduje volanie `brk`, ktoré nastavuje veľkosť dátového segmentu – v každom výpise `strace` ho

nájdeme hneď niekoľkokrát. Nasledujú dva pokusy o otvorenie súboru. Obidva má na starosti dynamický linker. Prvý `open` sa snaží otvoriť súbor `/etc/ld.so.preload`, kam je možné uvádzať knižnice načítané takpovediac z donútenia. Tento súbor však na danom počítači neexistuje a je vidieť, že volanie vrátilo hodnotu −1, ktorá značí neúspech s chybou ENOENT a popisom, z ktorého je jasné, že súbor nebol nájdený. Pretože to pre program nie je kritická chyba, pokračuje ďalej. Program otvorí súbor `/etc/ld.so.cache`, kde sú uložené informácie o dynamicky linkovateľných knižniciach v prehľadávaných adresároch. Linker tak zistí, kde sú uložené knižnice nutné k behu programu. Vrátené kladné číslo je identifikátor otvoreného súboru (takzvaný file descriptor), ktorý je hneď použitý v ďalšom volaní `fstat64` ako prvý argument. Toto volanie vracia informácie o súbore. Ďalšie volanie namapuje obsah súboru do pamäte. Volanie `close` zavrie otvorený súbor určený identifikátorom, ktorý vrátilo volanie `open`. Pri otváraní súboru sa vždy použije najnižší možný kladný celočíselný identifikátor, takže je možné, že pri opakovanom otvorení ďalšieho súboru môže volanie `open` vrátiť rovnaký identifikátor. Nasleduje opäť volanie `open`, ktoré tentokrát otvorí základnú knižnicu `libc`. Ako je z výpisu vidieť, telo programu, ako zamýšľal programátor sa ešte stále nedostalo na rad, vo všeobcenosti platí, že najprv si dynamický linker pootvára súbory, ktoré potrebuje, po ňom vetšinou nasleduje inicializácia `libc` (nastavenie lokalizácie, otvorenie katalógov správ, atď.) a až na koniec príde na rad funkcia main. V našom prípade s programom `false` bolo zavolaných zhruba 80 systémových volaní, z ktorých iba jediné a posledné plní vlastnú funkciu programu, ktorou je ukončenie programu s návratovým kódom rôznym od nuly – `_exit(1)`.

3.3 Ďalšie programy

3.3.1 Dtrace

`DTrace` [1] je komplexný dynamický sledovací program vytvorený firmou Sun Microsystems pre riešenie problémov jadra a aplikačných problémov na systémoch bežiacich v reálnom čase. Bol pôvodne vyvinutý pre Solaris, od tej doby je pod CDDL licenciou a bol portovaný na niekoľko ďalších Unixových systémov. `DTrace` môže byť použitý na získanie celkového prehľadu bežiaceho systému, ako je veľkosť pamäte, čas CPU, súborový systém a sieťové zdroje, ktoré používajú aktívne procesy. Môže tiež poskytnúť oveľa presnejšie informácie, napríklad log z argumentov, s ktorými je špecifická funkcia volaná, alebo zoznam procesov, ktoré pristupovali ku konkrétnemu súboru.

3.3.2 Truss

Utilita `truss` [8] spustí zadaný príkaz a stopuje, aké systémové volania tento príkaz vykonáva, aké signály obdrží, a taktiež zobrazuje chyby, ktoré sa vyskytnú. Každý riadok výstupu zobrazuje chyby, mená signálov alebo mená systémových volaní aj s argumentami a návratovou hodnotou. Argumenty systémových volaní, ak je to možné, sú zobrazené symbolicky, použitím definícií z relevantných systémových hlavičiek. Pre každý argument, ktorý obsahuje ukazateľ na cestu k súboru, sa táto cesta zobrazí vo výstupe ako hodnota string. Pri chybových návratových hodnotách je uvedená aj hodnota premennej `errno` pre ľahšiu identifikáciu chyby.

3.3.3 Ktrace

`Ktrace` [2] je nástroj, ktorý je súčasťou niektorých verzií BSD Unix a Mac OS X a dokáže sledovať interakciu jadra z programami a zapisovať toto sledovanie na disk pre neskoršiu analýzu. Je veľmi podobný Linuxovému nástroju `strace`, ale narozdiel od neho je omnoho rýchlejší. Pri nástroji `strace`, pre každé systémové volanie, ktoré je uskutočnené programom, je potrebné prepnúť kontext na sledovací program a späť, narozdiel od `ktrace`, kde je sledovanie uskutočňované jadrom, takže nie je potrebné ďalšie prepínanie kontextu a záznamy o volaniach sú ukladané v banálnej podobe. Log súbory vytvorené pomocou `ktrace` sa dajú prezerať pomocou nástroja `kdump`. V operačnom systéme Mac OS X 10.5 Leopard bol `ktrace` nahradený nástrojom dtrace.

3.3.4 SystemTap

`SystemTap` [7] poskytuje zadarmo (GPL) softvérovú infraštruktúru na zjednodušenie zbierania informácií o bežiacom Linuxovom systéme. Táto infraštruktúra pomáha pri diagnostikovaní výkonnostného alebo funkcionálneho problému. `SystemTap` eliminuje potrebu developera prejsť zdĺhavým a nepohodlným procesom sekvencií rekompilovania, inštalovania a reštartovania, ktoré by inak boli potrebné pre zbieranie dát. `SystemTap` obsahuje jednoduché rozhranie príkazového riadka a skriptovací jazyk pre písanie inštrukcií pre bežiace jadro.

Technológie `dtrace` a `SystemTap` sú určené pre real-time monitorovanie produkčných systémov, keď umožňujú sledovať napríklad jednotlivé systémové volania len s minimálnym respektíve temer žiadnym negatívnym dopadom na výkon. Umožňujú tak analyzovať záťaž najmä serverových aplikácií v reálnom nasadení, optimalizovať aplikácie a systémy a objavovať príčiny limitov výkonu.

Veľkou výhodou nástroja `strace` oproti nástroju `SystemTap` je nekomplikovaná možnosť analyzovať beh programu viackrát. Túto vlastnosť budem využívať aj v mojej práci, kde bude možné vďaka tomu spraviť analýzu toho istého behu programu viackrát, a to dokonca aj pri zmenených pravidlách kontroly.

3.4 Experiment

Na záver tejto kapitoly som sa rozhodol spraviť 2 experimenty. V prvom porovnám, ako jednotlivé linuxové grafické prostredia využívajú systémové volania. Testované boli 3 distribúcie Ubuntu (Gnome), Kubuntu (KDE) a Xubuntu(Xfce) s Linuxovým jadrom 2.6.32-17. Beh programu bol sledovaný pomocou programu `strace`. Výstup programu `strace` som analyzoval a systémové volania rozdelil do 5 hlavných kategórií: správa procesov, správa súborov, správa zariadení, správa informácií a komunikácia. V druhom experimente analyzujem všetky nástroje a utility ktoré sa nachádzajú v adresári `/bin` a taktiež spravím štatistické porovnanie najčastejšie používaných systémových volaní v tomto adresári.

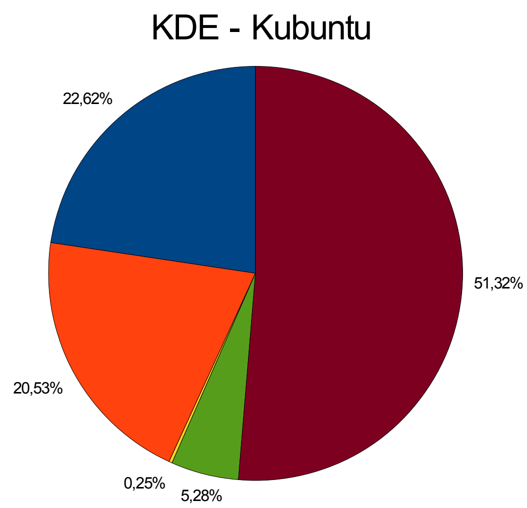
3.4.1 Testovací scenár experimentu č.1

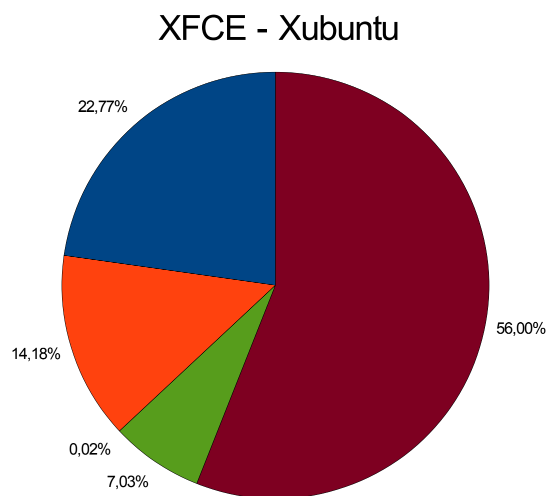
Sledovaný bol beh celého operačného systému od jeho spustenia po dobu 20 minút. Prvých 10 minút bolo vyhradených na prezeranie webu a počas zvyšných desiatich minút boli otvárané a uzatvárané rôzne dokumenty.

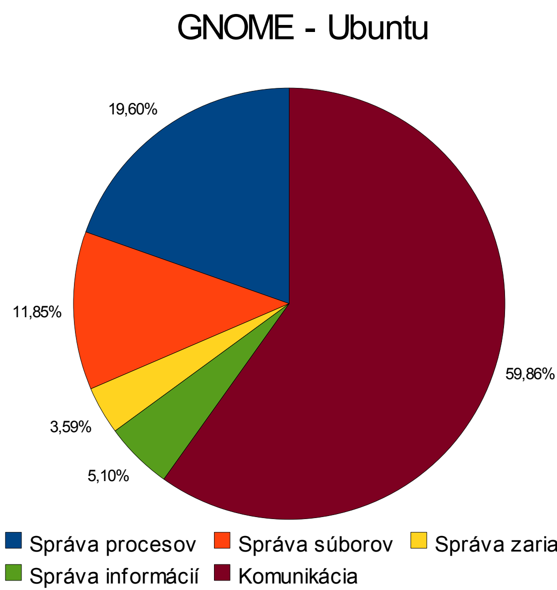
3.4.2 Výsledok experimentu č.1

Výsledok experimentu odhalil viacero zaujímavých informácií. Program strace zistil, že približne 6% zo všetkých systémových volaní vo všetkých 3 testovaných prostrediach sa skončili chybovou návratovou hodnotou. Skoro všetky chyby (až 90%) boli z kategórie pracujúcej zo súbormi. Jednalo sa však prevažne o takzvané false alarmy, ktoré nemali vplyv na beh programu. Išlo o uzatvorenie už zatvoreného súboru, otvorenie neexistujúceho súboru, pokus o čítanie zo súboru, keď boli zdroje dočasne nedostupné, kontrolu prístupových práv do súboru, ktorý neexistoval atď.

Obrázok 3.2 znázorňuje percentuálne rozloženie systémových volaní v piatich kategóriach. Experimentom som zistil, že aj keď sa jednalo o rôzne desktopové prostredia, rozloženie volaní bolo približne rovnaké a ich počet tiež. Za 20 minút sa na všetkých troch testovaných prostrediach vykonalo v priemere 2 800 000 volaní. Najviac ich bolo v prostredí Gnome (∼ 2 850 000), najmenej v prostredí Xfce (∼ 2 740 000). Tieto rozdiely však mohli byť spôsobené aj tým, že pri jednotlivých testoch boli napríklad prezerané iné webové stránky, ktoré mohli obsahovať viac/menej dát. Ako je z obrázkov vidieť, na všetkých troch prostrediach boli prevádzané najčastejšie systémové volania, ktoré mali na starosti komunikáciu, nasledovala správa procesov, správa súborov, správa informácií a na poslednom mieste sa umiestnila správa zariadení.







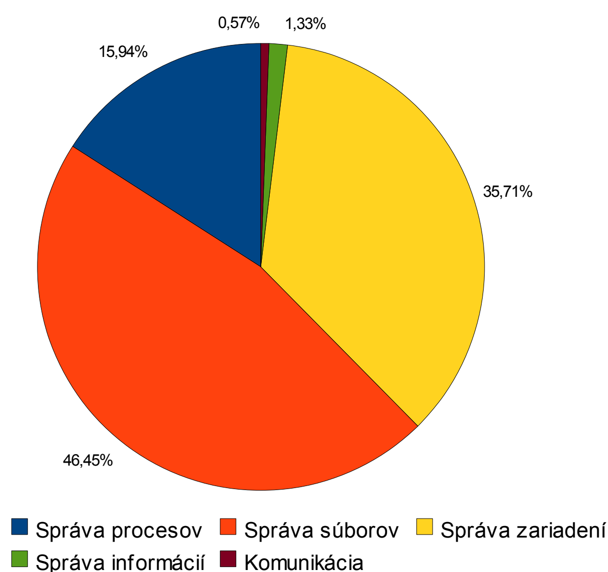
Obrázok 3.2: Rozdelenie systémových volaní v KDE, Gnome a Xfce.

3.4.3 Testovací scenár experimentu č.2

Experimentom s číslom 2 bolo odsledovanie všetkých programov v adresári `/bin` na školskom servery Merlin. Tento adresár obsahuje najviac potrebné programy, ktoré systém potrebuje, aby mohol správne fungovať, napríklad shell, príkazy ls, grep atp. V tomto adresári sa nachádzalo ku dňu 5.5.2010 107 programov. Všetky programy som pomocou nástroja strace analyzoval a opäť výsledok rozdelil do 5 kategórií tak, ako tomu bolo aj v prípade predchádzajúceho experimentu.

3.4.4 Výsledok experimentu č.2

Počas sledovania všetkých programov z adresára `/bin` sa vykonalo približne 96 tisíc systémových volaní. Najčastejším volaním sa stalo systémové volanie `ioctl` (∼ 34 000x), ktoré má na starosti správu zariadení. Na druhom a treťom mieste sa umiestnili volania read a write (∼ 13 000x a ∼ 9 000x). Ako môžete na obrázku 3.3 vidieť, takmer polovica všetkých systémových volaní patrí do kategórie, v ktorej sa pracuje zo súbormi (vytvorenie, vymazanie, otvorenie, zatvorenie, čítanie, zápis). Komunikácia narozdiel od predchádzajúceho experimentu v tomto prípade tvorila veľmi mizivé percento systémových volaní. Stalo sa tak predovšetkým kvôli tomu, že zo všetkých programov, ktoré sa nachádzajú v adresári `/bin` ich len zopár slúži na komunikáciu.



Obrázok 3.3: Rozdelenie použitých systémových volaní z adresára /bin.

Kapitola 4

Návrh programu

Základnou úlohou programu na kontrolu systémovývh volaní je vhodnou a užívateľovi \ testerovi zrozumiteľnou formou zobraziť údaje a informácie o behu programu. Výstup programu `strace` môže obsahovať tisícky záznamov o vykonaných systémových volaní a stáva sa tak veľmi ťažko čitateľný hlavne pre menej skúsených užívateľov systému Linux. Ako príklad môžem uviesť môj experiment, v ktorom som sledoval grafické prostredia Linuxu a v ktorých sa za 20 minút vykonalo takmer 3 milióny systémových volaní. Pri takomto čísle by mal aj skúsený užívateľ problém nájsť chybu medzi všetkými volaniami. Programom, ktorý sa snažím navrhnúť a implementovať by som chcel neskúseným, ale aj pokročilejším užívateľom zjednodušiť a sprehľadniť výpis systémových volaní, ktoré boli počas behu programu vykonané.

4.1 Špecifikácia požiadaviek

Špecifikácia požiadaviek prebiehala po dohode s vedúcim práce, Ing. Alešom Smrčkom. Samotné zadanie dáva už pomerne ucelené požiadavky a zoznam uvedený na nasledujúcich riadkoch zhrňuje všetky základné požiadavky na program:

- možnosť otestovať akýkoľvek program
  - navrhovaný program musí mať schopnosť otestovať akýkoľvek program bez ohľadu na množstvo prepínačov atp.
- jednoduché užívateľské rozhranie
  - užívateľ by mal byť schopný rýchlo sa zorientovať v rozhraní programu a dokázať ho ihneď po spustení používať.
- uložiť beh programu (výstup strace)
  - uloženie behu programu zabezpečí, že užívateľ bude schopný opakovane spúšťať analýzu toho istého programu a to dokonca aj s inými pravidlami.
- možnosť spustenia automatického testu
  - automatický test umožní kompletnú analýzu programu zo všetkých pravidiel, ktoré užívateľ vytvoril.

  - užívateľ má tak prehľad o kompletnom behu programu a nemusí sa k jednotlivým odpovediam na otázky popísané v pravidlách dopracovávať zadávaním jednotlivých otázok.

- možnosť zadávať jednotlivé pravidlá samostatne
  - užívateľ by mal mať taktiež možnosť zadávať jednotlivé otázky aj samostatne. Užívateľa totiž môže zaujímať len určitá časť programu a automatickým testom by stratil čas, keďže vyhodnotenie jednotlivých odpovedí, hlavne pri programoch, počas ktorých prebehlo veľké množstvo volaní by mohlo trvať dlhšiu dobu.

- možnosť zadávania otázok nad behom programu formou otázka-odpoveď
  - program by mal byť schopný prijímať aj také otázky, ktoré nemá popísané v pravidlách. Užívateľ v tomto prípade v príkazovom riadku vytvorí „pravidlo“ a program mu musí byť schopný na takéto pravidlo odpovedať.

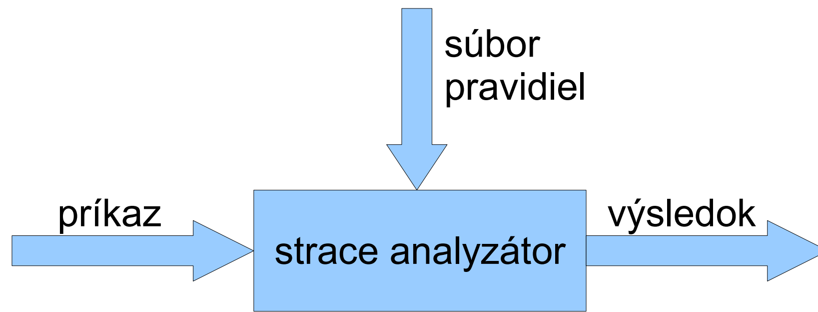
- možnosť napísať vlastné pravidlá, ktoré otestujú špecifickú časť programu
  - každý užívateľ si môže vytvoriť špecifické pravidlá, podľa ktorých chce testovať svoj program. Týmto krokom bude zabezpečená maximálna otvorenosť programu na kontrolu systémových volaní a program sa tak bude dať prispôsobiť pre potreby každého užívateľa.

- možnosť opakovanej analýzy rovnakého behu programu (rovnakého výstupu strace)
  - vďaka tomu, že vlastné pravidlá si bude môcť vytvoriť každý užívateľ, bude potrebné zaručiť, aby bolo možné ten istý beh programu otestovať opakovane. Užívateľ nemusí mať pri začatí testovania pripravené všetky pravidlá a tieto pravidlá vytvorí až v priebehu testovania a vďaka možnosti uloženia zoznamu vykonaných systémových volaní počas behu testovaného programu tak bude možné novo vytvorené pravidlá spustiť nad už uloženým behom programu.

4.2 Návrh testovacieho systému

Schéma testovacieho systému je vyobrazená na obrázku 4.1. Užívateľ zadá príkaz, ktorý chce skontrolovať, poprípade namiesto tohoto príkazu môžeme „poslať“ do analyzátora výstup v minulosti vytvoreného `strace-u`. Taktiež môže užívateľ vytvoriť ďalšie pravidlá, ktorými chce otestovať výstup programu `strace`. Tento analyzátor následne prevedie všetky úkony potrebné na to, aby užívateľ na výstup dostal informácie o všetkom, čo popísal v pravidlách. O tom, ako vytvoriť pravidlá sa dočítate v sekcií 6. Výsledkom je súbor odpovedí na otázky zadané pravidlami.

Aby bolo možné jednoduchým a nekomplikovaným spôsobom umožniť užívateľovi vytvárať a používať pravidlá, pomocou ktorých dokáže otestovať rôzne programy, bude potrebné použiť programovací jazyk v ktorom bude tieto pravidlá možné zapísať a umožní uživateľovi vyjadrovať sa logickým popisom úlohy, namiesto toho aby sa zaoberal písaním strohých počítačových inštrukcí a v programe určoval, čo všetko a ako sa má robiť. Bude potrebné vytvoriť takpovediac inteligentnú databázu znalostí o systémových volaniach a pomocou takéhoto jazyka určiť či sa počas behu sledovaného programu nevyskytla chyba.



Obrázok 4.1: Schéma testovacieho systému.

Kapitola 5

Implementácia programu

V tejto časti práce sa budem venovať implementácii mojej úlohy a jej problematike. Uvediem nástroje, ktoré boli pri implementácii použité a pokúsim sa priblížiť všetky implementačné detaily, ktorými som sa pri riešení práce musel zaoberať, aby som zaručil funkčnosť a korektné správanie programu.

5.1 Použité nástroje

Pre implementáciu programu som využil programovací jazyk Python [4]. Python je vysoko výkonný programovací jazyk používajúci efektívne vysokoúrovňové typy, pričom jednoducho a elegantne rieši otázku objektovo orientovaného programovania. Jeho syntax a dynamické typy spolu z interpretovaním kódu dotvára povesť ideálneho nástroja pre písanie skriptov a rýchly vývoj aplikácií. Samostatný interpret jazyka je spustiteľný na veľkom množstve platforiem vrátane Linuxu, Windows, MacOS a DOS. O Pythone sa ďalej tvrdí, že je vhodný pre výuku programovania, písania prototypov aplikácií, webových aplikácií, aplikácií typu klient/server, sieťových aplikácií, skriptov, utilít, systémových programov, menších grafických aplikácií, prácou s textovými dokumentami vrátane XML, skriptovania iných aplikácií a podobne. Je v ňom možné písať jednoduchšie i multithreadové aplikácie, používa sa pri paralelnom programovaní, je obľúbený v akademickej sfére, má dobrú podporu i pre Windows a existuje jeho implementácia aj pre prostredie .NET a Java. Python je vyvíjaný ako open source projekt, a je v súčasnost vo verzii 3.1.2 a 2.6.5.

Pravidlá dotazovacieho jazyka boli napísané v programovacom jazyku Prolog. Prolog je logický programovací jazyk. Patrí medzi tzv. deklaratívne programovacie jazyky, v ktorých programátor popisuje len cieľ výpočtu, pričom presný postup, akým sa k výsledku program dostane, je ponechaný na ľubovôli systému. Prolog sa snaží o pokiaľ možno abstraktné vyjadrenie faktov a logických vzťahov medzi nimi s potlačením imperatívnej zložky. Prolog je využívaný predovšetkým v obore umelej inteligencie a v počítačovej lingvistike (obzvlášť spracovanie prirodzeného jazyka, pre ktorý bol pôvodne navrhnutý). Syntax jazyka je veľmi jednoduchá a ľahko použiteľná práve preto, že bol pôvodne určený pre počítačovo nie príliš gramotných lingvistov. Prolog je založený na predikátovej logike prvého rádu

Ako interpret dotazovacieho jazyka bol využitý modul PySwip [3]. Vďaka licencii GNU GPL som mohol tento modul slobodne využiť pre použitie v mojej práci. PySwip modul slúži ako most, ktorý prepája SWI-Prolog s Pythonom a dovoľuje v programoch napísaných

v Pythone kladenie otázok nad databázami znalostí. PySwip v podstate zahrňuje rozhranie jazyka SWI-Prolog. PySwip používa SWI-Prolog ako zdieľanú knižnicu a ctypes na prístup k nej, takže nevyžaduje kompiláciu pred inštalovaním. Ctypes je rozhranie pre cudzie funkcie v jazyku Python, umožnuje volanie funkcií z DDL alebo zdielaných knižníc. Je to v podstate mechanizmus pomocou ktorého program napísaný v jednom jazyku, može volať rutiny napísane v inom jazyku.

5.2 Program

Programová časť tejto sa skladá z dvoch častí. Tou prvou je rozparsovanie výstupu `strace` a jeho spracovanie do dopredu stanovenej formy a tou druhou je samostatné vyhodnotenie tohto výstupu.

5.2.1 Analyzovanie a parsing výstupu strace

Aby bolo možné sledovať, ktoré systémové volania boli volané, v akom čase a s akými parametrami, bolo nevyhnuté výstup z programu strace prerobiť do podoby, ktorej bude interpret dotazovacieho jazyka rozumieť. Rozhodol som sa prerobiť tento výstup do syntaxe jazyka Prolog, keďže budem v programe používať aj modul, ktorý dokáže pracovať s pravidlami tohto jazyka. Každé jedno systémové volanie bolo prevedené do nasledovnej podoby:

```
názov_volania(čas_od_spustenia,PID,návratová_hodnota,arg1,arg2,arg3,...).
```

Ako príklad ešte uvediem ukážku konkrétneho systémové volania z výstupu `strace` a jeho podobu, ktorej bude Prolog rozumieť a bude ju schopný spracovať:

```
2488 1271250876.949666 open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
==>
open(000.00492900, 2488, 3, mapa1, 1).
```

Z predchádzajúcej ukážky môžeme vidieť, že niektoré argumenty neodpovedajú uvedenému systémovému volaniu. Dôvod je veľmi jednoduchý – napríklad na to, aby sme mohli pracovať so súbormi, bude treba zisťovať s akými právami bol súbor otváraný a keďže existujú iba 3 varianty otvorenia súboru (iba zápis, iba čítanie, zápis aj čítanie), rozhodol som sa argumenty `O_RDONLY, O_WRONLY, O_RDWR` nahradiť číselnými konštantami, kde číslo 1 značí, že išlo o otvorenie súboru na čítanie, číslo 2 značí, že sa súbor otvoril aj na čítanie, aj na zápis a číslo 3 značí, že išlo o otvorenie súboru iba na zápis. Taktiež bola ošetrená situácia, kde niektoré systémové volanie bolo zavolané, ale skončilo stavom nedokončené (`unfinished`) a jeho ukončenie sa vykonalo až niekedy v budúcnosti:

```
2227 write(1, "c += 2 :  2\n", 11 <unfinished ...>
2227 <...write resumed> ) = 11
```

Ďalším krokom bolo nahradenie všetkých znakov, ktoré sa vo výstupe strace nachádzali medzi znakmi hranatých zátvoriek, množinových zátvoriek a úvodzoviek. K tomuto kroku som pristúpil preto, lebo sa medzi týmito znakmi veľmi často vyskytoval znak apostrofu, v ktorom sa v syntaxy Prologu nachádzajú reťazce. Aby neprišlo k strate niektorých

dôležitých údajov, ktoré sa nachádzajú medzi takýmito znakmi (napríklad aký súbor sa otvoril), tak som tento problém vyriešil jednoduchým spôsobom: takýto reťazec som uložil do slovníka Pythonu, v ktorom kľúč k tomuto reťazcu bolo nahradené slovo. Takže v tomto prípade by sme sa k reťazcu „`libc.so.6`“ dostali zadaním kľúča `mapa1`. Posledným krokom, ktorý som musel spraviť, aby som neporušil syntax jazyka Prolog bolo vloženie všetkých argumentov začínajúcich na veľké písmeno, do úvodzoviek, keďže v jazyku Prolog sa veľkým písmenom začínajú premenné. Čas systémového volania vo výstupe nezodpovedá času odpovedajúceho volania z výstupu programu `strace`. Je to preto, lebo program `strace` vypisuje čas volania ako číslo v sekundách od polnoci 1.januára 1970 a výstup môjho volania obsahuje čas, ktorý prešiel od spustenia sledovaného príkazu.

Pri používaní nástroja `strace` som zistil, že hlavne pri jednoduchých príkazoch sa za tú istú milióntinu sekundy vykonalo viacero systémových volaní. Tieto prípady som musel taktiež ošetriť, keďže budem pracovať z časovou logikou a interpret, ktorý budem používať, bude očakávať unikátnu hodnotu času pre každé jedno systémové volanie. Taktiež som objavil niektoré volania, ktoré majú premenlivý počet parametrov (systémové volania `clone`, `close`, `sysinfo`, `fcntl`, `fcntl64`, `futex`). Tieto volania sú tiež spracované, ale boli odstránené všetky ich parametre, takže obsahujú len čas, v ktorom boli volané, ich identifikačné číslo procesu a návratovú hodnotu, s ktorou boli skončené. Ďalším problémom boli volania, ktoré sa začínajú znakom _. Keďže syntax jazyka Prolog nepovoľuje, aby sa pravidlo začínalo znakom _, tak som pomocou regulárnych výrazov pred tento znak vložil znak s. Takže napríklad pre systémové volanie `_llseek`, bude tomuto volaniu v databáze znalostí odpovedať volanie `s_llseek`. Nakoniec som pri systémových volaniach ktoré skončili chybou, vytvoril ďalší slovník, kde ako kľúč bol čas, v ktorom k tejto chybe došlo a hodnota bol chybový kód s popisom chyby.

5.2.2 Vyhodnotenie výstupu strace

Druhá časť programu je zameraná na interpret dotazovacieho jazyka a zaoberá sa vyhodnotením výstupu `strace`. Pomocou modulu PySwip a výstupu `strace`, ktorý bol spracovaný v prvej časti programu sa vytvorí databáza znalostí o všetkých systémových volaniach, ktoré boli volané počas behu programu a následne pomocou zadávania dotazov nad touto databázov sa vyhodnotí, či sa v programe nenachádzala chyba, ktorú sme popísali v pravidlách. Program umožňuje zadávanie dotazov samostatne, cez konzolu, alebo je možné spustiť automatický test, kde sa zo súboru načítajú všetky pravidlá obsiahnuté v tomto súbore, vyhodnotia sa a následne sa na výstup alebo do súboru zapíše výsledok týchto pravidiel. Program ešte predtým ako odpoveď vypíše do súboru alebo na štandardný výstup, skontroluje, či sa v tomto výstupe nenachádza kľúč k slovníku, a ak sa nachádza tak tento kľúč nahradí za hodnotu, ktorá patrí danému kľúču. Taktiež sa výpis abecedne zoradí a odstránia sa duplicitné výskyty.

Zadávanie dotazov je veľmi jednoduché a riadi sa syntaxov jazyka Prolog. Ak by sme chceli zistiť, ktoré súbory otvorilo systémové volanie `open`, stačilo by zadať do konzoly nasledovný príkaz:

```
open(_,_,_,Subor,_).
```

A výsledkom by bol výpis všetkých súborov, ktore boli počas behu programu otvorené

(aj neúspešne otvorené subory). Znak podtržítka označuje, že údaj, ktorý je na tomto mieste nás nezaujíma. Ak by sme chceli vedieť aj čas, v ktorom volanie open otvorilo súbory, tak by sme vyššie uvedený príkaz upravili do podoby:

```
open(Cas,_,_,Subor,_).
```

Vo všeobecnosti je možné „odsledovať“ akékoľvek systémové volanie, avšak je potrebné, aby interpret poznal počet jeho parametrov, ktoré je možné zistiť napríklad z manuálových stránok jednotlivých volaní.

5.2.3 Interaktívny mód

V programe bol implementovaný aj interaktívny mód zadávania otázok nad databázou znalostí. Tento mód je aktívny vždy, keď užívateľ nespustí program s parametrom spustenia automatizovaného testu. Mód funguje na princípe otázka-odpoveď a po každej otázke, ktorú užívateľ zadá do príkazového riadku sa snaží pomocou znalostí a pravidiel ktoré už obsahuje zistiť odpoveď na zadanú otázku. Pri neúspešnom hľadaní program oznámi užívateľovi pomocou odpovede `false`, že v databáze znalostí nenašiel odpoveď. Ak by užívateľ zadal neplatnú otázku, tiež mu bude táto skutočnosť oznámená.

Kapitola 6

Pravidlá a jazyk

Ako som už spomínal v predchádzajúcich kapitolách, výstup programu závisí od vytvorených pravidiel. V tejto kapitole sa bližšie pozrieme na to ako tieto pravidlá vyzerajú a popíšeme si spôsob, ako takéto pravidlá môžeme vytvoriť.

6.1 Ukážka pravidla

```
spocitaj_precitane(TIME,SPOLU):-
  retractall(counter(_)),
  assert(counter(0)),
  assert(read(0,0,0,0,0,0)),
  spocitaj_read(TIME),
  retract(counter(SPOLU)).
spocitaj_read(TIME):-
  read(TIME1,_,X,_,_,_), TIME1>TIME,X>0, retract(counter(OLD)),
  NEW is OLD + X, assert(counter(NEW)),fail; spocitaj_readv(TIME).
```

Toto pravidlo slúži na zistenie, koľko dát bolo počas behu programu spracovaných systémovým volaním `read`. Prvá časť pravidla vynuluje počítadlo, pridá do databázy znalostí jedno volanie `read` a zavolá ďalšie pravidlo, ktoré prejde cez všetky volania `read` a postupne inkrementuje počítadlo vždy o počet bytov, ktoré volanie `read` obsahuje v návratovej hodnote.

Ak by sme takéto pravidlo chceli zahrnúť aj do automatizovaného testu, stačí, ak hocikam do súboru, ktorý obsahuje takéto pravidlo dopíšeme riadok s nasledovnou syntaxov:

```
%%spocitaj_precitane(0,SPOLU).
```

Ak by sme chceli zahrnúť do výstupu aj popisok, či informáciu, čo aktuálne pravidlo znamená, stačí pridať pred vyššie uvedený riadok ešte jeden so syntaxov:

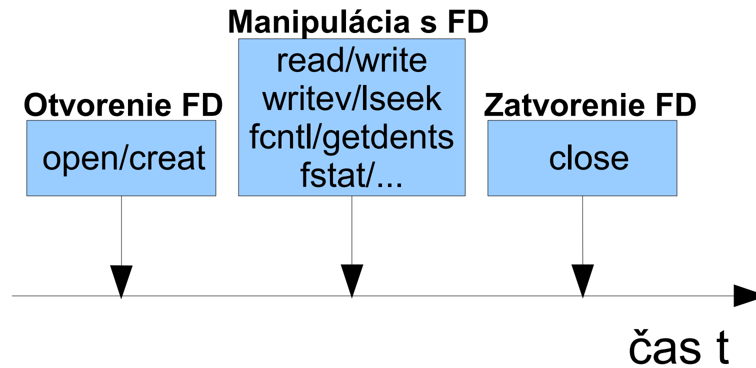
```
%%_Vypise pocet spracovanych dat systemovym volanim read
```

Vo všeobecnosti platí pre každý riadok v súbore s pravidlami začínajúci znakmi %% bude tento riadok interpretovaný ako pravidlo, ktoré sa má pri automatizovanom teste

spustiť a riadok začínajúci znakmi %%_ bude označovať popis takéhoto pravidla. Týmto krokom som zabezpečil, že výpis bude prehľadnejší a užívateľ, ktorý spustil automatizovaný test bude vedieť, čo kontrolovali jednotlivé pravidlá.

6.2 Vytvorenie všeobecného pravidla

V tejto časti práce sa pokúsim vytvoriť univerzálne pravidlo, ktoré by bolo aplikovateľné aj na ďalšie pravidlá. Aby som bol presnejší, bude sa jednať o pravidlo, ktoré bude kontrolovať, či sa nejaké systémové volanie vykonalo až po tom, ako sa vykonalo iné systémové volanie, ale pred tým, ako sa vykonalo ďalšie. V podstate budeme pracovať len s časovými údajmi, v ktorých budeme porovnávať, či časy jednotlivých volaní prebehli v takom poradí, v akom sme si ich určili. Zadanú situáciu môžete vidieť aj na obrázku 6.1, na ktorom je znázornená následnosť systémových volaní pri manipulácií s file descriptorom (file descriptor musí byť najprv otvorený, potom s ním môžeme manipulovať a nakoniec by mal byť zavretý).



Obrázok 6.1: Následnosť systémových volaní.

Ak zoberiem do úvahy nasledujúci kód:

```
universal_rule(T,argumenty):-
  sys_call_1(T,argumenty),
  sys_call_2(T2,argumenty),
  T2>T, sys_call_3(T3,argumenty),
  T3>T, !,
  T2<T3 -> (assert(counter(argumenty_ktore_chceme_vo_vypise)),
      universal_rule(Tx,argumenty) ; true) ;
  universal_rule(Tx,argumenty);
  true.
```

Takéto pravidlo znamená, že Prolog nájde v databáze znalostí hľadané `sys_call_1` s časom `T`, potom nájde také `sys_call_2`, ktorého čas je väčší ako čas `sys_call_1` a následne nájde prvé `sys_call_3`, ktorého čas je väčší ako čas `sys_call_1`. Porovná časy `sys_call_2`

a `sys_call_3`, a ak čas druhého systémového volania je menší ako čas tretieho, znamená to, že sa nám v programe vyskytlo systémové volanie na mieste, kde sme ho neočakávali (ako príklad uvediem systémove volanie `read`, medzi volaniami `open`, ktoré bolo volané s parametrom pre zápis do súboru a volaním `close`). Ak takéto volanie nájdeme, tak sa vloží do takzvaného počítadla (`counter`), v ktorom sa ukladajú všetky nájdené výskyty a rekurzívne sa zavolá tá istá funkcia, aby sme docielili prehľadanie celej databázy znalostí.

Aby sme naozaj docielili prehľadanie celej databázy znalostí, je potrebné vedieť časy prvého volania, podľa ktorého sa bude zisťovať následnosť druhého a tretieho. Tento spôsob docielime jednoduchou inicializáciou, ktorú zavoláme ešte pred samotným spustením prehľadávania.

```
init_universal_rule(T,argumenty):- retractall(counter(_,_)),
  findall(Time,sys_call1(Time,_,_,_,3),Zoznam),
  universal_rule(Zoznam,argumenty); counter(argumenty_ktore_budu_vo_vypise).
```

Pomocou tejto inicializácie nájdeme všetky časy systémového volania 1, ktoré potom predáme pravidlu `universal_rule`. Následne v tomto pravidle ešte zmeníme hlavičku do podoby `universal_rule([T|Telo],argumenty)` a pravidlo budeme rekurzívne volať pomocou `universal_rule(Telo,argumenty)`.

Univerzálnym pravidlom, aké som vytvoril na predchádzajúcich riadkoch, by sa dalo vytvoriť mnoho podobných pravidiel, ktoré by kontrolovali neexistujúce file descriptory, práva na prístup do súborov, chyby vzniknuté pri komunikácii cez sockety atď., stačí len porozmýšľať, čo chceme cez systémové volania skontrolovať, prepísať univerzálne pravidlo tak, aby vyhovovalo potrebám systémových volaní, ktoré chceme cez pravidlo vyhľadať a som presvedčený o tom, že sa vám to pomocou takéhoto pravidla úspešne podarí.

Kapitola 7

Záver

Testovanie sa často dostáva niekam do úzadia. Zrejme je to preto, že si celkom neuvedomujeme jeho dôležitosť. Niektoré menej závažné chyby (napríklad pomalší beh programu) nám používateľ môže „odpustiť“. Ale v prípade systémov, ktoré by v dôsledku chyby mohli spôsobiť finančné alebo materiálne škody, prípadne ohroziť zdravie alebo dokonca životy ľudí, je nevyhnutné, aby sa k testovaniu pristupovalo naozaj zodpovedne. Jedine zodpovedný prístup môže viesť ku kvalitnému výrobku, ktorý spokojný zákazník ocení tým, že v budúcnosti opäť využije naše služby.

Cieľom mojej práce bolo vytvoriť jednoduchú aplikáciu, ktorá by bola schopná skontrolovať akýkoľvek program a pomocou zadaných pravidiel zistiť, či počas jeho behu nedošlo k závažným chybám. Domnievam sa, že požadovaný cieľ som splnil, aj keď by sa dalo v tejto práci ešte mnoho vecí vylepšiť. Hlavnú úlohu zohral aj čas, ktorý mi neumožnil vytvorenie toľkých pravidiel, koľko som mal prvotne v pláne a svoju úlohu zohral aj fakt, že sa táto problematika na internete príliš nerozoberá a bolo veľmi obtiažne vyhľadať relevantné informácie. Pri opakovanom spracovaní tejto práce by som pravdepodobne postupoval trochu inak, aj keď základné princípy by určite zostali zachované. Pri analýze systémových volaní som sa naučil mnoho nových vecí a uvedomil som si zložitosť koncepcie linuxového jadra. Skript, ktorý som vytvoril si môže nájsť uplatnenie najmä u začínajúcich programátorov a študentov, ktorí pri tvorbe programov spravia chyby a namiesto zložitého hľadania vo výpisoch debuggerov či nástrojov zobrazujúcich systémové volania, môžu tento skript použiť na objavenie chyby, ktorú či už z nepozornosti alebo nevedomosti spravili. Pravidlá, ktoré som spolu so skriptom vytvoril slúžia hlavne na demonštráciu činnosti tejto práce a sú zamerané prevažne na kontrolu operácií so súbormi, avšak pomocou týchto pravidiel by sa dalo vytvoriť mnoho ďalších, ktoré by mohli kontrolovať napríklad systémové volania používané pri komunikácií, vytváraní nových procesov a podobne.

Literatúra

[1] Dtrace manual pages[online].
http://compute.cnr.berkeley.edu/cgi-bin/man-cgi?dtrace+7, [cit. 2010-5-4].

[2] Ktrace manual pages[online]. http://www.manpagez.com/man/1/ktrace/, [cit. 2010-5-4].

[3] Pyswip homepage[online]. http://code.google.com/p/pyswip/, [cit. 2010-5-4].

[4] Python tutorial [online]. http://docs.python.org/tutorial/, [cit. 2010-5-4].

[5] Spôsoby testovania [online]. http://www.cleverandsmart.cz/zpusoby-testovani/, [cit. 2010-5-4].

[6] Strace tutorial [online]. http://linux.die.net/man/1/strace, [cit. 2010-5-4].

[7] SystemTap [online]. http://sourceware.org/systemtap/, [cit. 2010-5-4].

[8] Truss manual pages[online].
http://fuse4bsd.creo.hu/localcgi/man-cgi.cgi?truss+1, [cit. 2010-5-4].

[9] Wikipedia – The Free Encyclopedia: Software testing [online].
http://en.wikipedia.org/wiki/Software_testing, [rev. 2010-5-1][cit. 2010-5-4].

[10] Manjarrez, J.: Implementing Linux System Calls [online].
http://www.linuxjournal.com/article/3326, [cit. 2010-5-4].

[11] Marick, B.: *The Craft Of Software Testing, Subsystem Testing*. Prentice Hall PTR, 1995, iSBN 0-13-177411-5.

[12] Mayers, G.: *The Art of Software Testing*. Wiley, New York, 1979, iSBN 13-978-0471043287.

[13] P. Hejl, J. T.; Klapetek, P.: Validace softwaru pro metrologii[online].
http://www.automatizace.cz/download.php?d=QXRtX0FydGljbGUscGRmX2FydCw4ODI=, [cit. 2010-5-4].

[14] Shah, H.: *On Run-Time Enforcement of Policies*. Bachelor thesis, Informatica & TLC, University of Trento, Italy, 2008.

Dodatok A

Príklad testu

Pre demonštrašné účely ukážem vytvorenie testu

Popis testu	kontrola programu ls
Vsupy	`.\analyzer.py −−c ‘‘sh -c \‘‘ls | wc -l\’’ ’’ −−auto result.out`
Očakávaný výsledok	program analyzuje zadaný príkaz a do súboru result.out zapíše výsledky popísané v pravidlách

```
==> Vypise pocet spracovanych a pocet zapisanych dat v kB <==
spocitaj_vsetky(0,Prijate,Odoslane)
Odoslane:  0.0258662762323
Prijate:  1.25683260127

==> Vypise vsetky otvorene subory <==
open(_,_,_,Subor,_)
Subor:  .
Subor:  /etc/ld.so.cache
Subor:  /lib/libacl.so.1
Subor:  /lib/libattr.so.1
Subor:  /lib/libselinux.so.1
Subor:  /lib/tls/i686/cmov/libc.so.6
Subor:  /lib/tls/i686/cmov/libdl.so.2
Subor:  /lib/tls/i686/cmov/libpthread.so.0
Subor:  /lib/tls/i686/cmov/librt.so.1
Subor:  /proc/filesystems
Subor:  /usr/lib/gconv/gconv-modules.cache
Subor:  /usr/lib/locale/locale-archive
Subor:  /usr/lib/locale/sk_SK.utf8/LC_ADDRESS
Subor:  /usr/lib/locale/sk_SK.utf8/LC_COLLATE
Subor:  /usr/lib/locale/sk_SK.utf8/LC_CTYPE
Subor:  /usr/lib/locale/sk_SK.utf8/LC_IDENTIFICATION
Subor:  /usr/lib/locale/sk_SK.utf8/LC_MEASUREMENT
Subor:  /usr/lib/locale/sk_SK.utf8/LC_MESSAGES
```

```
Subor:  /usr/lib/locale/sk_SK.utf8/LC_MESSAGES/SYS_LC_MESSAGES
Subor:  /usr/lib/locale/sk_SK.utf8/LC_MONETARY
Subor:  /usr/lib/locale/sk_SK.utf8/LC_NAME
Subor:  /usr/lib/locale/sk_SK.utf8/LC_NUMERIC
Subor:  /usr/lib/locale/sk_SK.utf8/LC_PAPER
Subor:  /usr/lib/locale/sk_SK.utf8/LC_TELEPHONE
Subor:  /usr/lib/locale/sk_SK.utf8/LC_TIME
Subor:  /usr/share/locale-langpack/sk.utf8/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale-langpack/sk/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale-langpack/sk_SK.utf8/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale-langpack/sk_SK/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale/locale.alias
Subor:  /usr/share/locale/sk.utf8/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale/sk/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale/sk_SK.utf8/LC_MESSAGES/coreutils.mo
Subor:  /usr/share/locale/sk_SK/LC_MESSAGES/coreutils.mo

==> Vypise neuspesne otvorene subory, ich cas a dovod <==
open(_Cas_a_dovod,_,-1,Subor,_)
Subor:  /usr/lib/locale/locale-archive
_Cas_a_dovod:  0.084637 - ENOENT (No such file or directory)
Subor:  /usr/lib/locale/locale-archive
_Cas_a_dovod:  0.098706 - ENOENT (No such file or directory)
Subor:  /usr/share/locale-langpack/sk.utf8/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.142119 - ENOENT (No such file or directory)
Subor:  /usr/share/locale-langpack/sk_SK.utf8/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.139312 - ENOENT (No such file or directory)
Subor:  /usr/share/locale-langpack/sk_SK/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.140762 - ENOENT (No such file or directory)
Subor:  /usr/share/locale/sk.utf8/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.136648 - ENOENT (No such file or directory)
Subor:  /usr/share/locale/sk/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.13781 - ENOENT (No such file or directory)
Subor:  /usr/share/locale/sk_SK.utf8/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.134292 - ENOENT (No such file or directory)
Subor:  /usr/share/locale/sk_SK/LC_MESSAGES/coreutils.mo
_Cas_a_dovod:  0.135614 - ENOENT (No such file or directory)

==> Vypise celkovy cas behu programu <==
doba_behu(T)
T: 0.172959

==> Vypise subory, kt.  boli otvorene na zapis a pokusalo sa z nich
citat <==
check_read(Subor,Time_Read)
false
```

```
==> Vypise subory, kt.  boli otvorene na citanie a pokusalo sa do nich
zapisovat <==
check_write(Subor,Time_Write)
false

==> Spocita pocet vetveni programu (volania fork a clone) <==
fork_clone(Count)
Count:  2
```

Ako môžeme z predchádzajúceho výpisu vidieť, beh príkazu `sh -c` „`ls | wc -l`“, ktorý vypisuje počet položiek nachádzajúcich sa v aktuálnom adresáre bol korektný. Nezapisovalo sa do žiadneho súboru otvoreného na čítanie a naopak. Z výpisu je taktiež možné vidieť, že beh programu sa ďalej rozdelil na 2 časti pomocou volania `fork`. Ďalej z výpisu môžeme vidieť, že sa nepodarilo otvoriť súbor `/usr/lib/locale/locale-archive`, ktorý však na testovanom stroji neexistuje a lokálne nastavenia sa nachádzajú v inej zložke. Ďaľším otváraným súborom, s ktorým mal testovaný počítač problém bol súbor `coreutils.mo`. Z výpisu môžeme vidieť, že sa tento súbor pokúšal systém nájsť v rôznych zložkách:
`/usr/share/locale/sk_SK.utf8/`,
`/usr/share/locale/sk_SK/`,
`/usr/share/locale/sk.utf8/`
a nakoniec ho našiel v zložke `/usr/share/locale-langpack/sk/LC_MESSAGES/`.

Dodatok B

Obsah CD

Na CD sa nachádzajú nasledujúce prílohy:

- bakalarka.pdf – bakalárska práca vo formáte pdf
- analyzer.py – skript určený na kontrolu programu
- pravidla.pl – súbor pravidiel na kotrolu programu
- INSTALL – manuál nastavení pre zabezpečenie funkčnosti skriptu
- zložka pyswip – modul využívaný skriptom pre komunikáciu s prologom
- zložka bakalarka – zdrojový tvar bakalárskej práce