VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ANALÝZA VLIVU PORUCH A CHYB NA CHOVÁNÍ JÁDRA OPERAČNÍHO SYSTÉMU

ON ANALYSIS OF FAULT AND ERROR IMPACT TO BEHAVIOR OF AN OPERATING SYSTEM KERNEL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADRIÁNA BLAŠKOVÁ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JOSEF STRNADEL, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19429/2016/xblask02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Blašková Adriána**
Obor: Informační technologie
Téma: **Analýza vlivu poruch a chyb na chování jádra operačního systému**
On Analysis of Fault and Error Impact to Behavior of an Operating System Kernel
Kategorie: Operační systémy

Pokyny:

1. Zdokumentujte základní pojmy z oblasti operačních systémů (OS) a vytvořte přehled OS pro dnešní platformy.
2. Po dohodě s vedoucím zvolte OS a platformu, se kterými budete dále pracovat - kritériem volby nechť je dostupnost. Přehledně a názorně zdokumentujte rozhraní jádra zvoleného OS a zvolte části jádra, které jsou z hlediska poskytování služeb nejdůležitější.
3. Klasifikujte typy poruch a chyb, vytvořte přehled mechanismů zvýšení spolehlivosti systémů; klasifikujte vlivy poruch a chyb na chování jader OS a aplikací jimi řízených.
4. Navrhněte a poté proveďte sadu experimentů, pomocí nichž bude možno klasifikovat vliv poruch a chyb na poskytování služeb jádra zvoleného OS a chod aplikace nad ním běžící. Nejprve studujte vliv samostatných poruch/chyb, poté se zaměřte i na jejich kombinace.
5. Výsledky shrňte a navrhněte mechanismy potlačení vlivů poruch či chyb.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Strnadel Josef, Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Cieľom tejto práce je analyzovať vplyv porúch a chýb na chod jadra operačného systému. Obsahuje podrobne spracované základné pojmy a techniky ako spoľahlivosť systému, poruchy a chyby alebo mechanizmy na zvýšenie spoľahlivosti. Zameriava sa taktiež na návrh, implementáciu, riadenie a vyhodnotenie experimentov za účelom klasifikovať vplyvy porúch na jadro vybraného operačného systému ale aj na aplikácie, ktoré nad jadrom bežia.

Abstract

The aim of this bachelor thesis is to analyze the impact of faults and errors on behavior of an operating system kernel. It contains detailed basic concepts and techniques such as system reliability, faults and errors or mechanisms for reliability increase. It focuses on design, implementation, execution and evaluation of experiments in order to classify the effects of faults to the selected operating system kernel but also to the applications, which are running on the kernel.

Kľúčové slová

operačné systémy, jadro, spoľahlivosť, systémy odolné proti poruchám, injektácia porúch, redundancia

Keywords

operating systems, kernel, reliability, fault tolerant systems, fault injection, redundancy

Citácia

BLAŠKOVÁ, Adriána. *Analýza vlivu poruch a chyb na chování jádra operačního systému.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Strnadel Josef.

Analýza vlivu poruch a chyb na chování jádra operačního systému

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracovala samostatne pod vedením pána Ing. Josefa Strnadela, Ph.D. Uviedla som všetky literárne pramene a publikácie, z ktorých som čerpala.

.....
Adriána Blašková
16. mája 2017

Poďakovanie

Týmto by som rada poďakovala vedúcemu mojej bakalárskej práce pánovi Ing. Josefovi Strnadelovi, Ph.D. za všetok venovaný čas, trpezlivosť a cenné rady.

Obsah

Kapitola 1

Úvod

Počítače majú v súčasnej dobe obrovské zastúpenie v rôznych sférach priemyslu, v zdravotníctve, či v školstve a podobne. Okrem klasického pohľadu na ne, sa počítače nachádzajú v automobiloch, zabezpečovacích systémoch, zaznamenávajú počasie alebo monitorujú zdravie človeka a iné. Práve preto sa od nich vyžaduje maximálna presnosť a správnosť jednotlivých úkonov a operácií. Každá porucha v počítači alebo celkovo v systéme (počítač s operačným systémom) môže mať veľmi citeľné následky. Kvôli tomu je dôležité aby chyba bola včas zaznamenaná a systém adekvátne zareagoval. Na to sa využívajú mechanizmy na zvýšenie spoľahlivosti, ktoré sú predmetom tejto práce.

Nasledujúca kapitola sa zameriava na objasnenie základných pojmov operačných systémov (OS) a obsahuje malý prehľad pre dnešné platformy podľa vybranej klasifikácie. Všeobecné princípy budú následne bližšie vysvetlené na ilustračnom príklade operačného systému Linux, ktorý bol využitý v druhej časti tejto práce. Použitý bol aj FreeRTOS, ktorého princípy sú uvedené v kapitole 4, avšak pre potreby kapitoly 2 bol vybraný práve populárnejší Linux.

V rámci kapitoly 3 bude čitateľ kompletne oboznámený s teoretickým základom témy, od klasifikácie porúch a chýb, ktoré sa môžu vyskytnúť v systéme, až po podrobný prehľad mechanizmov na zvýšenie spoľahlivosti systému. Príkladom sú rôzne druhy redundancií alebo kontrolných techník pre detekciu chýb.

Druhá (praktická) časť práce je venovaná experimentom na klasifikáciu vplyvu porúch a chýb na chod jadra OS a aplikácií nad ním bežiace. Návrh a spôsob implementácie jednotlivých experimentov sú uvedené v kapitole 4. Princípom je umelo zavádzať metódou *fault injection* poruchy do systému a vyhodnotiť ich vplyv.

Vyhodnotenie simulácie je možné nájsť v kapitole 5, formou textového vysvetlenia ale aj grafickej podoby (viz. Príloha) pre lepšiu orientáciu. Taktiež je tu možné nájsť zhodnotenie vykonaných experimentov a návrh na zvýšenie spoľahlivosti systémov.

Kapitola 2

Základné pojmy v oblasti operačných systémov

Sú dva základné pojmy v oblasti operačných systémov, a to samotný **operačný systém** a **jadro operačného systému**. Od týchto dvoch zložiek sa ďalej odvíjajú ich vlastnosti, funkcie alebo kritériá, podľa ktorých je ich možné ďalej klasifikovať.

V tejto kapitole sú všeobecne vysvetlené dané pojmy a vytvorený malý prehľad systémov a jadier podľa vybranej klasifikácie. Všeobecné princípy sú potom podrobnejšie ilustrované na konkrétnych príkladoch, ktorými sú operačný systém Linux a jeho jadro.

2.1 Operačný systém

V dnešnej dobe sa počítačové (výpočtové) systémy skladajú z niekoľkých komponentov. V prvom rade to je procesor, ďalej určitá pamäť, disky, rôzne vstupno-výstupné periférne zariadenia ako napríklad klávesnica, monitor, tlačiareň, a podobne. Na to, aby užívatelia počítačových systémov mohli s týmito zložkami korektne pracovať, je vytvorená softvérová úroveň, nazývaná *operačný systém* (OS). Všeobecne sa dá *operačný systém* popísať ako program, respektíve skupina programov, ktoré prispôsobujú technické prostriedky stanoveným typom aplikácií a požadovanému režimu činnosti počítača [10]. Inak povedané, táto základná zložka programového vybavenia spája užívateľov a ich aplikácie s hardvérom výpočtového systému, ktorý môže byť taktiež virtualizovaný.

Za dva základné ciele OS sa považujú, v čo najväčšej miere využitie dostupných zdrojov daného počítača a vytvorenie jednoduchého a intuitívneho prostredia pre užívateľa. Od tohto sa odvíjajú aj jeho funkcie a to:

- spravovať a riadiť zariarenia a zdroje
- vytvoriť užívateľské prostredie s danou mierou abstrakcie nad prostriedkami

Operačný systém je zvyčajne chápaný tak, že sa skladá z jadra, (viz sekcia 2.2), systémových knižníc a užívateľského rozhrania [19]. V odborných literatúrach sú uvedené viaceré klasifikácie OS podľa rôznych parametrov, ako napríklad podľa úrovne zdieľania procesora, typu interakcie, podľa miery distribuovateľnosti, počtu užívateľov alebo tiež podľa vnútornej štruktúry OS a tak ďalej.

Pre ukážku sú vybrané tri typy OS s uvedenými príkladmi (podľa [16]):

- Desktopové OS (Personal Computer OS) - Microsoft Windows, Linux, Mac OS X
- Systémy reálneho času (Real-Time OS) - FreeRTOS, VxWorks, QNX
- Vnorené systémy (Embedded OS) - PalmOS, Windows CE

2.1.1 Operačný systém Linux

Pod pojmom Linux sa označuje počítačový operačný systém, ktorý začal tvoriť v roku 1991 fínsky programátor Linus Torvalds ako náhradu komerčného systému Minix. Po vytvorení prvej funkčnej verzie, sprístupnil zdrojový kód na internete. Termínom Linux (vzniklo spojením slov Linus a Unix) bolo najskôr označované len jadro, ale postupom času sa takto začala označovať komplexná platforma, vrátane grafických prostredí a množstva programov. V súčastnosti je Linux plne vybavený operačný systém, ktorý obsahuje grafické rozhranie ako napríklad KDE, Gnome a podobne, aplikácie pre prácu s internetom, s grafikou ale aj bežné kancelárske balíčky, multimediálne prehrávače či hry. Je to multiužívateľský sieťový operačný systém, ktorý sa používa ako 32 aj ako 64-bitová verzia s podporou multitaskingu[1]. Keďže Linux je voľne šíriteľný mnohí užívatelia resp. spoločnosti vytvorili tzv. distribúcie, ktoré majú spoločný základný systém tvorený jadrom s ovládačmi zariadenia a typy programov podľa zamerania danej distribúcie. Ďalej sa vyznačujú inštalačným programom, štýlom nastavenia a podobne. Medzi najznámejšie distribúcie Linuxu patria Ubuntu, Debian, Fedora alebo Linux Mint a podobne. Hlavná výhoda Linuxu je jeho *flexibilita*. Tým sa myslí, že môže pracovať na rôznych hardvérových platformách od superpočítačov cez notebooky až po napríklad sieťové karty. Ďalšou významnou výhodou je, že Linux dokonale spĺňuje štandard POSIX [7], vďaka čomu je možná jednoduchá tvorba programov prenositeľných medzi unixovými systémami.

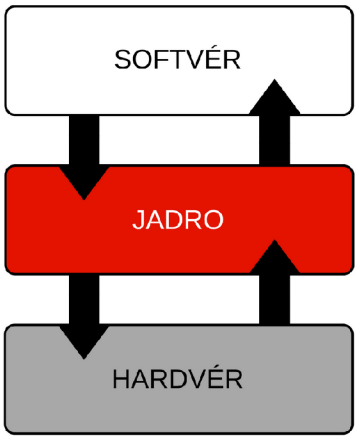
2.2 Jadro operačného systému

Centrálnym prvkom väčšiny operačných systémov je určitý správca systému, ktorý je nazývaný *jadro OS*. To sa zavádza do operačnej pamäte už pri štarte počítačového systému a je mu priradené riadenie. Najčastejšie tiež koordinuje činnosť počítača po celú dobu jeho behu. Je to zároveň najnižšia a najzákladnejšia časť operačných systémov, ktorá obvykle beží v privilégovanom režime [19]. Hlavnými úlohami *jadra OS* je správa prostriedkov počítačového systém a vytváranie takého prostredia, ktoré zabezpečí správnu činnosť vyšším vrstvám operačného systému, ako je možné vidieť na obrázku 2.1. Jadro je teda priamo spojené s hardvérom systému, taktiež virtualizovaným, a tým riadi prideľovanie pamäte a času procesora programom (zodpovedá za to časť **plánovač**), ovládanie diskových jednotiek, správu periférnych zariadení a podobne [25]. Výhodou *jadra OS* je, okrem iného, zapuzdrenie hardvéru daného počítačového systému, čo umožňuje spusteným programom prístup k prostriedkom bez poznania ich špecifikácií. Ďalšími podstatnými funkciami *jadra* sú napríklad ochrana dát na disku, celková bezpečnosť systému, spravuje sieťovú komunikáciu ale aj multitasking.

[1] Multitasking jadra je podstate ilúzia, ktorá sa užívateľovi javí ako beh viacerých nezávislých programov naraz.

Podľa toho aké služby konkrétne *jadro* ponúka, ho môžme zaradiť do jednej z týchto typov *jadier OS* (uvedené aj s príkladmi) [19]:

- **monolitické jadrá** - vytvárajú komplexné a efektívne prostredie, napr. FreeBSD, Linux
- **mikrojadrá** - obmedzujú rozsah jadra na jednoduché rozhranie, príklad mikrojadra: QNX alebo Mach
- **hybridné jadrá** - rozšírené mikrojadrá, napr. Mac OS X, Windows NT
- **exojadrá** - experimentálne jadrá, napr. Aegis, Nemesis



Obr. 2.1: Funckia jadra operačného systému.

2.2.1 Jadro operačného systému Linux

Ako bolo už uvedené v predchádzajúcej kapitole, Linux je prevažne monolitické jadro ale niekedy, záleží od danej architektúry, môže byť považovaný za hybridný typ jadra [7]. Monolitický typ jadier sa vyznačuje hlavne tým, že celý kód beží v rovnakej pamäťovej oblasti, tzv. *kernel space*. Teda všetky služby operačného systému sú spolu s hlavným vláknom v privilegovanom režime, tým sa myslí napríklad správa pamäti, podpora sieťovej komunikácie, plánovanie a podobne. Takýto model bol vytvorený za účelom vysokej efektivity najmä, čo sa týka prístupu k hardvéru. Vytvára vysokoúrovňové komplexné rozhranie s veľkým počtom služieb a vysokou abstrakciou, ktorú ponúka nadradeným vrstvám. Keďže sú jednotlivé časti jadra veľmi úzko previazané, jeho najväčšou nevýhodou je práve táto závislosť, keďže chyba v jednom subsystéme alebo ovládači môže zablokovať alebo dokonca zhodiť celý systém. To rieši vylepšený typ jadier tzv. monolitické jadro s modulárnou štruktúrou, kde je možné spravovať jednotlivé subsystémy v podobe modulov za behu [19].

Kapitola 3

Mechanizmy na zvýšenie spoľahlivosti systémov

Na to, aby bolo možné hodnotiť a porovnávať spoľahlivosť určitého systému je nutné, aby boli definované veličiny, v ktorých sa spoľahlivosť bude udávať. Aj keď spoľahlivosť ako takú dokáže každý užívateľ systému popísať, nie je sama o sebe klasifikovateľná [13]. Podľa technickej normy ČSN 010102 je charakterizovaná ako:

> *„všeobecná vlastnosť objektu, ktorá spočíva v schopnosti plniť požadované funkcie tak, aby boli zachované hodnoty stanovených prevádzkových ukazateľov, ako napríklad rýchlosť, spotreba energie a pod., v daných medziach a v čase, podľa zadaných technických podmienok.“*

Táto komplexná vlastnosť zahrňuje bezporuchovosť, udržateľnosť, životnosť, zaistenie údržby a podobne. Vysvetlenie pojmov, prebrané z [17]:

- **Bezporuchovosť:** je schopnosť objektu nepretržite vykonávať požadované funkcie v stanovenú dobu a za daných podmienok
- **Opraviteľnosť:** v prípade vzniku poruchy objektu, zistiť príčinu a odstránenie následnou opravou
- **Udržovateľnosť:** prevencia proti poruchám pravidelnou údržbou
- **Životnosť:** schopnosť objektu plniť požadované funkcie do dosiahnutia krajného stavu, tým sa myslí stav, v ktorom je prerušená ďalšia činnosť objektu
- **Pohotovosť:** komplexná vlastnosť, ktorá zahŕňa bezporuchovosť a opravitelnoť v podmienkach prevádzky

3.1 Poruchy a chyby systému

Pri štúdií spoľahlivosti sú veľmi dôležité termíny *porucha* a *chyba*. Často sú v rôznych literatúrach zmieňované a nerozlišované. V tomto dokumente bude *porucha* (anglicky *fault*) predstavovať hardvérovú alebo softvérovú (programovú) vadu. Inak povedané porucha je jav, kedy už systém nemá schopnosť plniť požadované funkcie podľa technických podmienok. Naopak *chyba* (anglicky *error*) je rozdiel medzi správnou a skutočnou hodnotou určitej veličiny, ktorú zistíme pozorovaním alebo meraním. To znamená, že chyba je prejavom

nejakej poruchy ale nie každá porucha sa môže prejaviť ako chyba [8]. Dobrým príkladom je, ak sa počas behu systému nepoužije porušená súčiastka.

Z tohto kontextu sa dajú definovať dva stavy systému a to stav *bezporuchový* (porucha nenastala) a stav *poruchový* (porucha nastala).

Jedna zo základných klasifikácií porúch a chýb je podľa úrovne systému, kde sa nachádzajú. Napríklad v rámci najnižšej úrovne sa môžu vyskytnúť poruchy elektrických vlastností súčiastok. Ďalej sa môžu prejaviť vady spôsobené zlou aplikáciou komponentov alebo nedodržaním ich technických podmienok a parametrov. Typickými príkladmi týchto porúch je skrat, prerušenie vodivých ciest a podobne. V rámci tejto práce budú analyzované poruchy a chyby na úrovni jadra operačného systému, ktoré často rieši práve daný operačný systém. Príkladom je nesprávna práca s pamäťovým subsystémom, vyčerpanie zdrojov alebo preťaženie systému. Za najvyššiu úroveň sa dá považovať užívateľský priestor, kde za zdroj chýb sa považuje samotný užívateľ [15].

3.1.1 Klasifikácia porúch a chýb

Poruchy a chyby môžu byť klasifikované do viacerých kategórií, záleží aké kritérium alebo veličina je skúmaná. Takým základným rozdelením je do dvoch tried. Prvé sú **softvérové poruchy**, do ktorých sa rátajú všetky dizajnové poruchy (poruchy vzniknuté už pri navrhovaní alebo programovaní systému), ďalej programové zlyhanie zapríčinené chybami v zdrojovom kóde a podobne. Potom sú tu **hardvérové poruchy**, kam patria poruchy súčiastok systému.

Poruchy môžu byť skúmané podľa [8] z niekoľkých aspektov.
Podľa ich dĺžky trvania:

- *Trvalé poruchy* (angl. *permanent faults*) - nastáva pri trvalom porušení komponentu. Tento stav pretrváva, kým nie je objekt vymenený alebo opravený.
- *Prechodné poruchy* (angl. *transient faults*) - v tomto prípade súčiastka sa stáva poruchové iba v určitý čas. Po danej dobe je komponet opäť funkčný.
- *Ojedinelé poruchy* (angl. *intermittent faults*) - striedanie medzi poruchovým a bezporuchovým stavom. Je to opakovaný výskyt prechodnej poruchy.

Podľa ich vývoja a dopadu na objekty:

- *Neškodná porucha* (angl. *benign fault*) - spôsobuje úplné poškodenie komponentu. Pri takýchto typoch porúch sa presne vie, ktorá súčiastka je chybná a je ju nutné opraviť, či nahradiť.
- *Zradná porucha* (angl. *malicious fault*) - takýmto druhom sa rozumie porucha, ktorá priamo nespôsobuje zrútenie súčiastky ale narúša jej správny chod. Napríklad vytvára rozumne vyzerajúce avšak chybné výsledky. Tieto poruchy sa ťažko hľadajú a tým aj ťažko odstraňujú.

Rozdelenie podľa toho, čo/kto poruchy zapríčinil je možné definovať dvomi triedami [3]:

- *Fyzické poruchy* (angl. *Physical faults*) - myslia sa tým najmä hardvérové poruchy, ktoré môžu byť zapríčinené fyzickými javmi z vnútornej časti systému, ako napríklad zmena prahového napätia, alebo javmi ovplyvňujúce systém a pochádzajúce z okolia, napríklad prostredie ako také, rôzne druhy vibrácií alebo elektromagnetické vlny.

- *Poruchy spôsobené ľuďmi* (angl. *Human-made fault*) - sú také, ktoré zapríčinil svojim konaním človek. Môžu to byť buď dizajnérske poruchy, ktoré vznikli pri návrhu systému, pri jeho modifikáciách alebo vytvorením zlých pracovných postupov. Ďalej sa sem zaraďujú chyby, ktoré boli spôsobené porušením práve technických návodov alebo parametrov systému, či konkrétnych súčiastiek. Často sa tieto poruchy zaraďujú aj do softvérových.

3.2 Odolnosť proti systémovým poruchám

V súčasnosti odhaliť poruchy v systémoch alebo aplikáciách, ktoré sa týkajú hlavne bezpečnosti a riadenia kritických situácií napríklad v automobiloch, vlakoch alebo lietadlách, je stále viac dôležitejšie. Nevýhodou sa stáva najmä čoraz väčšia komplexnosť a zložitosť systémov, čím je garancia prijateľnej úrovne spoľahlivosti komplikovanejšia. Vysoká dostupnosť je nevyhnutne potrebná pre každý počítačový systém, ktorý je priamo spojený s ľudskou bezpečnosťou alebo ekonomickými investíciami [18].

Aby bolo možné zabezpečiť spoľahlivosť systému používajú sa rôzne mechanizmy na zvýšenie odolnosti proti systémovým poruchám. Táto vlastnosť systému mu umožňuje vykonávať úlohy aj v prípade, že nastane porucha, či už softvérových alebo hardvérových komponentov. Najčastejšou metódou na zvýšenie odolnosti je *redundancia*.

3.3 Redundancia v systéme

Redundancia je vlastnosť mať viac prostriedkov alebo zdrojov ako je minimálne nevyhnutné pre funkčnosť. V prípade, ak by nastala porucha v systéme, redundancia je využívaná na maskovanie a poskytuje ďalšie riešenie týchto nedostatkov, teda zachováva požadovanú funkčnosť [8]. V bezchybnom prostredí je redundancia takpovediac zbytočná. Môže to byť napríklad duplikácia hardvérového komponentu, časť v zdrojom kóde na kontrolu výpočtu alebo pridanie kontrolného bitu do daného registru.

Existujú štyri základné druhy redundancie, ktoré budú definované a rozobrané v nasledujúcich podkapitolách, a to *hardvérová, softvérová, informačná* a *časová*.

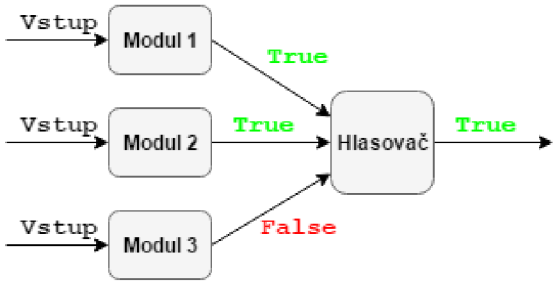
3.3.1 Hardvérová redundancia

Tento druh redundancie sa zaisťuje tým, že sa do systému pridá extra hardvérová súčiastka na rozpoznanie alebo nahradenie chybového dielu ale aj kombinovaním zložitých zdvojených štruktúr, ktoré sa stanú nadradenými, ak sa aktívna štruktúra dostane do poruchy. Príkladom môžu byť pridanie ďalšieho procesora, napájacieho zdroja, pamäte a podobne. V niektorých situáciách sa môže stať, že hardvérová redundancia je najlepšia možnosť ako zvýšiť spoľahlivosť systému, keďže ďalšie varianty ako kvalitnejšie komponenty, ladenie softvéru alebo kontrola kvality sú drahšie, komplikovanejšie alebo náročnejšie na údržbu. Pri duplikovaní jednotlivých súčiastok sa používa komparácia výsledkov alebo hlasovanie k zistení prítomnosti poruchy.

Tento druh redundancie je možné rozdeliť do troch ďalších podkategórií na základe obmedzenej aktivity modulov:

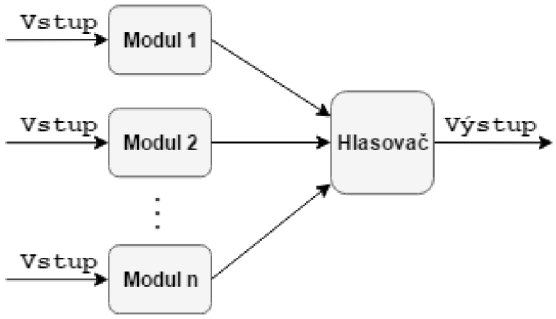
Statická (pasívna)

Táto metóda redundancie je tiež známa ako metóda "*maskovania*", pretože navyše pridané komponenty sa využívajú na maskovanie porúch v rámci daného hardvérového modulu. Výstup z modulu zostáva bez zmeny, tj. bez náznaku poruchy, pokým je ochrana účinná. Podstatou tejto techniky je, že redundantné kópie prvkov sú trvalo napájané a všetky vykonávajú rovnaké funkcie (čiže paralelne). V praxi sa využívajú rôzne formy statickej redundancie ako napríklad jednoduchá replikácia vybraných komponentov. Oveľa častejšia je metóda **trojitej modulárnej redundancie (TMR)** (obrázok 3.1), kedy dané komponenty sú strojené, ich výsledky sú následne vyhodnocované v tzv. *hlasovači*, ktorý majoritnou metódou vyberie správny výsledok, preto je maskovanie poruchy automatické a okamžité.



Obr. 3.1: Trojitá modulárna redundancia - majoritná metóda.

Podľa princípov **TMR** bola vytvorená ďalšia forma statickej redundancie s názvom **N-násobná modulárna redundancia (NMR)** (obrázok 3.2), kedy namiesto trojice rovnakých komponentov je ich počet n-násobný. Číslo n býva spravidla nepárne aby *hlasovač* mohol majoritnou metódou zvoliť správny výsledok operácie. Táto metóda redundancie sa ale v praxi nevyužíva často kvôli vysokým nákladom.



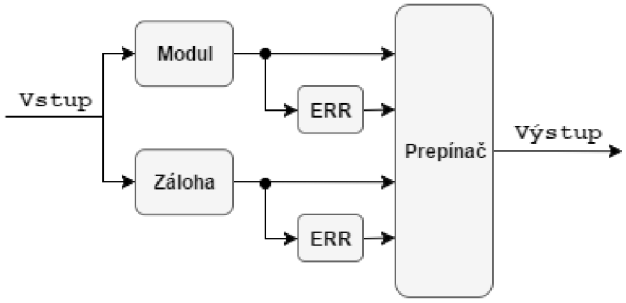
Obr. 3.2: N-násobná modulárna redundancia.

Avšak, ak porucha je nemaskovateľná a spôsobí chybu, oneskorené obnovenie alebo oprava nie je možná. Statická redundancia sa využíva proti prechodným aj trvalým poruchám. Príkladom môže byť použitie viacerých procesorov.

Dynamická (aktívna)

Dynamická hardvérová redundancia sa vyznačuje detekciou chyby, teda chybového signálu (vzniknutej poruchy) až na výstupe modulu. Odolnosť voči poruchám je implementovaná

v dvoch krokoch, najskôr sa detekuje vzniknutá porucha a potom nastáva obnovenie, kedy sa buď chybný komponent odstráni alebo opraví. To znamená, že náhradné komponenty sú aktivované počas zlyhania aktuálne využívaného modulu, teda lokalizuje postihnuté časti a následne ich odpája zo systému. Túto operáciu, ako aj vyhodnocovanie a monitorovanie vzniknutých porúch vykonáva modul - *prepínač*, čo je zobrazené na schéme 3.3. Táto technika (zvyčajne s podporou softvérovej a časovej redundancie) poskytuje tzv. "samoopravu" počítačového systému. Typickým príkladom je použitie parity na zistenie chyby pri prenose a ukladaní dát. Dynamická redundancia je využívaná pri systémoch náchylných na dočasné poruchy.



Obr. 3.3: Dynamická (aktívna) redundancia.

Podľa [4] je možné túto formu redundancie rozdeliť do ďalších dvoch kategórií a to:

- **Horúca záloha** (angl. *Hot standby sparing*) - pri tomto type je napájaný okrem aktuálne bežiaceho modulu aj záložný, aby v prípade poruchy operačného modulu, mohol byť záložný okamžite aktivovaný a následne chybný komponent odpojený zo systému.
- **Studená záloha** (angl. *Cold standby sparing*) - sa vyznačuje tým, že záložný modul je aktivovaný až po zistení poruchy u aktuálne bežiaceho komponentu. Nevyhodou sa stáva potrebný čas na uvedenie modulu do operačného stavu, na jeho inicializáciu a tiež následné opakované vykonanie danej operácie. Naopak výhodou je nižšie spotrebné náklady ako pri horúcej zálohe, pretože náhradné komponenty nie je potreba trvalo napájať.

Hybridná

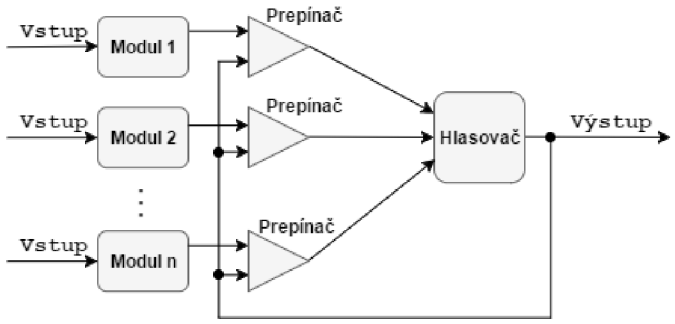
Hybridnou hardvérovou redundanciou sa rozumie kombinácia statickej a dynamickej redundancie. Maskovanie chýb sa využíva k oprave okamžitých chybných výsledkov a pomocou dynamickej redundancie sú chyby lokalizované.

Tieto formy hardvérovej redundancie vynakladajú vysoké režijne nároky, a preto sa zvyčajne používajú pre kritické systémy, kde je tento efekt druhoradý. Najčastejšie sa používajú ako ochrana pred *zradnými poruchami*.

Dobrým príkladom na vysvetlenie hybridnej formy je **Systém s automatickým čistením** (angl. *Self-purging redundancy*). Ako je už pri hardvérovej redundancií známe, aj v tomto prípade, systém obsahuje viaceré zhodné moduly, ktoré majú aktívny podiel na hlasovaní o správnom výsledku danej operácie (obrázok 3.4). Následne je výstup hlasovača porovnávaný s individuálnymi výstupmi jednotlivých modulov pre detekciu chýb. Keď je

nájdená nezhoda, daný chybový modul je prepínačom "vyčistený"(odpojený) zo systému. To znamená, že vstup takejto komponenty je nastavený na nulu a tak sa nemôže zúčastniť na hlasovaní. V takomto systéme je komponenta *hlasovač* prahové hradlo, ktoré má schopnosť prispôsobiť sa rôznym počtom vstupov.

Za predpokladu, že systém obsahuje n počet vstupov, je schopný maskovať $n-2$ chybných komponentov. V prípade, že v systéme sa nachádzajú iba dva bezchybové moduly, teda nastalo $n-2$ porúch, hlasovač nebude schopný rozhodnúť, ktorý výsledok je správny.



Obr. 3.4: Systém redundancie s automatickým čistením.

3.3.2 Softvérová redundancia

Softvérová odolnosť voči poruchám je technika navrhnutá tak, aby systém toleroval poruchy softvéru, ktoré tam zostali od jeho vyvíjania. Je možné tvrdiť, že každý softvérový systém, ktorý bol kedy vyrobený obsahuje chyby, čo býva najčastejšie spôsobené nekvalitným návrhom. Softvérová redundancia zahŕňa všetky dodatočné programy, programové segmenty, inštrukcie a mikroinštrukcie, ktoré sú potrebné pre bezporuchový stav počítačového systému. Veľa takýchto metód sa snaží využívať pre detekciu chýb techniky hardvérovej redundancie. Avšak hardvérová redundancia poskytuje dostatočnú odolnosť voči trvalým poruchám komponentov, na chyby návrhu nie sú primerane schopné.

Návrhové a implementačné chyby nemožno zistiť jednoduchou replikáciou identických softvérových štruktúr (ako je to pri hardvérovej redundancií), pretože by vznikla rovnaká chyba v každej kópií. Ak je rovnaký program kopírovaný a nastane zlyhanie v jednej z replík programu, bude táto porucha aj v ostatných kópiách, a to nie je spôsob ako vyriešiť problém. (Predpokladá sa rovnaký vstup pre každú repliku programu.) Jedným z riešením je, rovnakú úlohu implementovať rôznymi spôsobmi. Základom rozmanitosti variant je mať rovnaké špecifikácie na výsledný objekt [12].

Softvérovú redundanciu možno rozdeliť na dve skupiny:

1. Verzia single

Princípom tejto verzie je zlepšiť odolnosť voči poruchám jednotlivých softvérových modulov. Cieľom modifikácie softvérovej štruktúry je schopnosť chyby detekovať, izolovať a zabrániť ich šíreniu.

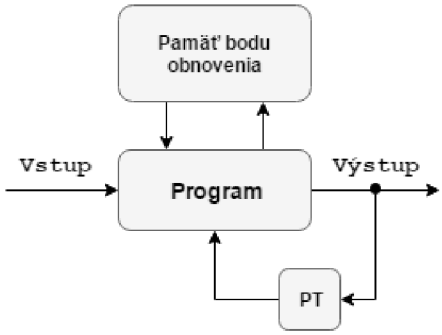
- **Techniky detekcie chýb** (angl. *Fault detection techniques*) - cieľom je rozpoznať, že nastala chyba v systéme a to napríklad vykonaním potvrdzujúcich testov. Výsledok programu je podrobený skúške, ak je správny, program pokračuje ak nie, je

nájdená chyba. Aby bol test čo najúčinnejší mal by byť ľahko spočítateľný a odvodený z kritérií nezávisle na aplikáciách. Potvrdzujúcimi testami sa rozumejú kontroly kódovania, spätné kontroly, časové a štrukturálne kontroly alebo kontrola primeranosti (viac v [4]).

- **Techniky izolácie chýb** (angl. *Fault containment techniques*) - je možné dosiahnuť úpravou systémovej štruktúry a vytvorením obmedzení na akcie, ktoré sú povolené v rámci systému. Do tejto kategórie patria techniky ako modularizácia, horizontálne alebo vertikálne delenie modulárnej hierarchie, uzavretie systému a tiež technika atomických akcií. Podstatou daných techník je identifikovať vzniknutú chybu vrámci modulu alebo určitej časti systému a následné obmedzenie komunikácie chybového modulu a ostatných častí. Obmedziť komunikáciu je možné zavedením jej presných pravidiel.

- **Techniky zotavenia z chýb** (angl. *Fault recovery techniques*) - akonáhle je detekovaná a izolovaná chyba, systém sa pokúsi zotaviť z chybového stavu a dostať sa späť do operačného. V prípade, že techniky detekcie a izolácie chýb sú implementované a nastavené správne, účinok poruchy sa preukáže počas detekcie danej chyby v rámci konkrétnej časti systému. Základné techniky zotavenia sú obsluha výnimiek, nastavenie bodu obnovenia a reštart, spracovanie párov alebo dátová diverzita. Veľmi často používanou technikou na zotavenie sa z chýb je práve obsluha výnimiek, kedy nastane prerušenie bežného operačného stavu systému kvôli zvládnutiu nepredvídateľnej a výnimočnej situácie.

  Ďalším rozšíreným mechanizmom je využitie bodu obnovenia a reštartovanie systému. Ako je možné vidieť na obrázku 3.5 technika je založená na princípe, kedy sa do pamäti bodu obnovenia uloží stav systému, buď pred začatím vykonávania operácie, alebo sa ukladá v priebehu výpočtu. Modul vykonáva operácie a jeho výsledky sú kontrolované pomocou **PT** - potvrdzujúcich testov. V prípade, že je detekovaná chyba, posiela sa signál do modulu na opätovnú inicializáciu stavu systému z pamäte bodu obnovenia.



Obr. 3.5: Technika vytvorenia bodu obnovenia a reštart.

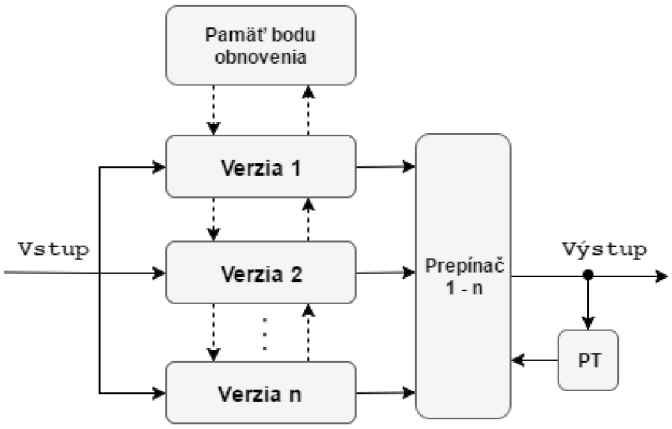
2. Verzia multi

Využívanie rôznych verzií modulu vytvorených na základe pravidiel diverzity.

Metódy softvérovej redundancie je možné použiť na jednotlivé moduly programov, celé aplikácie alebo kompletný systém. Je nutné ale poznamenať, že rovnako ako aj v hardvérových metódach na zvýšenie spoľahlivosti, aj v tomto prípade zvýšenie redundancie môže

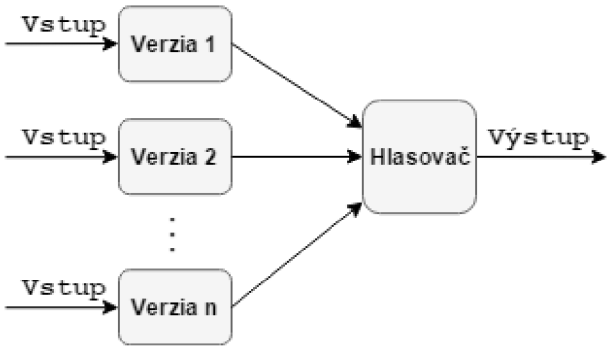
viesť ku komplikovanosti a degradovaní systému. Najčastejšie sa ale využívajú v letectve, pri použití nukleárnej energie, v zdravotníctve ale aj v iných priemyselných odvetviach [20].

- **Bloky zotavenia** (angl. *Recovery blocks*) - táto technika spojuje body obnovenia a reštart spolu s princípom záložnej redundancie (viz. 3.3.1 - Dynamická redundancia). Základné usporiadanie modulov je zobrazené schémou 3.6, kde body obnovenia sú vytvorené pred spustením programu. Verzie 1 až n predstavujú odlišné implementácie rovnakého programu ale iba jeden z nich sa podieľa na systémovom výstupe. Ak je potvrdzujúcimi testami odhalená chyba v momentálne aktívnej verzií, posiela sa signál o tomto stave do prepínača. Systém sa vráti do stavu uloženého v pamäti bodu obnovenia a prepínač predá riadenie ďalšej z verzií implementácie. V prípade, že všetky verzie sa dostanú do chybového stavu, prepínač vyvolá výnimku, ktorá tak informuje systém o zlyhaniu a neschopnosti dokončiť operáciu.



Obr. 3.6: Technika *mutli* - bloky zotavenia.

- **N-násobné programovanie** (angl. *N-version programming*) - technika je podobná N-násobnej modulárnej hardvérovej redundancie (NMR). Ako je možné vidieť na 3.7 systém tvorí n rôznych softvérových implementácií rovnakého programu, ktoré súbežne vykonávajú operáciu. Každá verzia pracuje s rovnakým vstupom ale iným spôsobom. Následne modul hlasovač, v tomto prípade *výberový algoritmus*, určí, ktorý z výsledkov je správny a zvolí ho na výstup. Existuje viacero typov hlasovačov. ktoré je možno v tejto technike využiť, ako napríklad klasický formalizovaný majoritný hlasovač, mediánový alebo technika výberu formalizovanou pluralitou a podobne.



Obr. 3.7: Technika *mutli* - N-násobné programovanie.

3.3.3 Informačná redundancia

Najznámejšou formou tohto typu odolnosti voči poruchám je detekcia chýb a korekcia pomocou kódovania, kde sú pridané extra bity (angl. *check bits*) do pôvodných dátových bitov, čo umožňuje overiť správnosť dát pred ich použitím, v niektorých prípadoch dokonca aj opravu chýb. Výsledná detekcia a korekcia chýb je v dnešnej dobe často využívaná v pamäťových jednotkách a v rôznych zariadeniach pre ukladanie dát ako ochrana proti neškodným poruchám. Pri využívaní informačnej redundancie sa vyžaduje taktiež pridanie ďalšej hardvérovej súčiastky na kontrolu bitov. Zavedenie informačnej redundancie prostredníctvom kódovania sa neobmedzuje len na úrovni jednotlivých dátových slov (angl. *data words*) ale môže byť rozšírená na odolnosť voči poruchám vo väčších dátových štruktúrach.

Táto technika sa radí medzi veľmi významné v oblasti prenášania dát, kde je nutná kontrola, či neprišlo k zmenám prenesených informácií. Oblasťou prenášania dát sa myslí, kedy sú dáta presúvané z jednej jednotky na druhú, z jedného systému do iného alebo aj ukladanie do pamäťovej jednotky počítačového systému.

Najznámejším príkladom takéhoto typu redundancie je RAID - *Redundant Array of Independent Disks* (doslova redundantné (nadbytočné) pole nezávislých diskov) a jeho rôzne variácie. Ďalšími bežne používanými metódami sú napríklad technika paritných kódov, lineárnych, cyklických kódov alebo Hammingov kód. Takýchto techník kódovania a ich verzií je na výber veľmi veľa, pre názornú ukážku budú dva z nich vysvetlené v nasledujúcom texte.

Paritný kód

Najjednoduchší a najstarší kód, ktorým je možno detekovať jednu chybu (angl. *Single Error Detection - SED*). Paritný kód je bežný binárny kód, kde ku kódu znaku je pridaný paritný bit tak, aby **parita** (parita počtu jednotiek) výsledného slova bola nulová. Párny (nepárny) paritný kód, ktorý je dĺžky n tvorený binárnu n-ticou obsahuje párny (nepárny) počet jednotiek. Detekcia jednobitovej chyby je možná pri porušení kódu teda párna parita sa stane nepárnou a naopak. Kombinácia so zvolenou párnou paritou sa označuje ako **kódová** a kombinácia s nepárnou ako **nekódová**.

Hammingov kód

Hammingov kód je jeden zo základných princípov konštrukcie binárneho kódu SEC, avšak je možné ho rozšíriť. Základnými parametrami kódu (n, k), kde n je celkový počet bitov v slove, vrátane m paritných bitov, takže $n = k + m$, pričom k určuje počet dátových bitov.

Hammingova vzdialenosť kódových zložiek je definovaná ako najmenší počet bitov, v ktorých sa dvojica kódových kombinácií odlišuje, čo sa zisťuje pre všetky možné kombinácie. Hammingova vzdialenosť teda medzi dvomi číslami x a y je určená $\delta(x, y)$, počtom bitových pozícií, v ktorých sú dané čísla odlišné. Príklad môže byť, ak $x = 0011$ a $y = 0101$, kde rozdiel je v 2 bitových pozíciách, teda $\delta(x, y) = 2$. Táto vzdialenosť udáva odhad počtu možných bitových chýb pri zmene x na y.

Hammingov kód je určený na opravu jednonásobných chýb za predpokladu pravdepodobnosti chyby $p < 0,5$. Táto technika je známa aj ako *perfektný kód*, ktorý má minimálnu možnosť redundancie. Jeho výhodou je aj pomerne rýchle a jednoduché kódovanie a dekódovanie. Požíva sa pri zabezpečení proti prechodným a trvalým chybám v RAM, zberníc a podobne.

3.3.4 Časová redundancia

Časová redundancia sa skladá z opakujúcich sa operácií, kedy každý jeden výsledok z predchádzajúcej operácie musí súhlasiť so súčasným výsledkom. Táto metóda sa môže uplatňovať na rôznych úrovniach systému ako napríklad mikrooperácie, jednotlivé inštrukcie, programové segmenty alebo celých programových štruktúr. Často sa využíva aj s inými technikami redundancie ako napríklad hardvérovou alebo softvérovou. Výhodou časovej redundancie je detekcia porúch a chýb, na základe opakovaného vykonávania rovnakej úlohy a potvrdenia jej výsledkov, ale aj možná obnova (oprava) daných chýb reštartovaním daného modulu. Pri tejto technike je teda nutné uloženie kópie predchádzajúceho výsledku operácie.

Časová redundancia sa využíva na rozlíšenie prechodných a trvalých porúch. To znamená, že v prípade ak sa chyba objaví iba v jednom či niekoľkých výsledkoch opakovaných operácií , nie všekých, ide o prechodnú chybu. Takto lokalizované komponenty, na ktorých sú postihnuté dočasnými poruchami, nie je nutné hneď odpájať zo systému.

3.4 Kontrolné body

Keď program potrebuje veľa operačného času na výpočet, pravdepodobnosť zlyhania počas vykonávania, ako aj náklady na ich opravu, sa stávajú veľmi vysoké. Aj napriek stále zvyšujúcej sa rýchlosti novodobých počítačov, sú známe viaceré dôležité aplikácie, ktoré potrebujú viac operačného času ako ostatné. Medzi ne patria napríklad simulácie zložitých fyzikálnych javov alebo matematických výrazov, komplexné optimalizácie alebo programy na podporu štúdií biochemických reakcií. V prípade poruchy pri vykonávaní takýchto zložitých operácií by po odstránení alebo opravení chyby, museli byť aplikácie spúšťané znova, čo zaberá mnoho času ale aj zvyšujúce sa nároky na výkon počítača. Tento problém je možné vyriešiť pomocou *Kontrolných bodov* (angl. *Checkpoints*).

Vo všeobecnosti platí, že *kontrolný bod* je snímka kompletného stavu procesu, v určitom momente, čo znamená, že musí obsahovať všetky informácie, ktoré sú potrebné na obnovenie procesu do tohto bodu. Vďaka tomu, pri poruche, sa systém vracia do najbližšieho kontrolného bodu (kedy bol systém ešte bezporuchový) a nevykonáva predošlé operácie znovu. Tieto kontrolné body je vhodné ukladať do stabilného pamäťového priestoru, kto-

rého spoľahlivosť je dostatočne overená. Najčastejším používaným pamäťovými médiami sú pevné disky, keďže dokážu uchovať dáta aj v prípade výpadku napájania [8]. Pri využívaní tejto metódy na zvýšenie spoľahlivosti systému je nutné dbať na:

- **Navýšenie kontrolnými bodmi** (angl. *checkpoint overhead*) - doba, o ktorú sa zvýši operačný čas aplikácie vytváraním kontrolných bodov
- **Latenciu kontrolných bodov** (angl. *checkpoint latency*) - čas potrebný na uloženie týchto bodov

3.4.1 Úrovne vytvárania kontrolných bodov

Vytváranie kontrolných bodov procesov je možne na rôznych systémových úrovniach [8].

- **Úroveň jadra** - vytváranie aj riadenie kontrolných bodov zostáva v jadre, procesy na vyšších úrovniach tieto operácie neovplyvňujú. V prípade ak nastane porucha, systém sa reštartuje a iba jadro je zodpovedné za riadenie obnovy.
- **Užívateľská úroveň** - v rámci tohto levelu je potrebné zabezpečiť potrebnú knižnicu, ktorá poskytuje vytváranie kontrolných bodov. Rovnako ako na úrovni jadra aj tu tento prístup nevyžaduje zmenu zdrojového kódu programu. Avšak je potrebné explicitné prepojenie s danou knižnicou na úrovni užívateľa. Táto knižnica taktiež riadi zotavenie po poruche.
- **Aplikačná úroveň** - aplikácia je zodpovedná za vytváranie a riadenie kontrolných bodov. Algoritmus pre tieto operácie musí byť implementovaný priamo do aplikácie. Táto metóda poskytuje užívateľovi najväčšiu kontrolu nad kontrolnými bodmi ale nevýhodou je nákladná implementácia a ladenie.

V podstate, každý moderný operačný systém poskytuje alebo používa kontrolné body. Ak je určitý proces prerušený, systém zaznamená a uchová jeho stav, takže výpočet môže pokračovať z bodu prerušenia bez toho aby nastala strata vykonaných operácií.

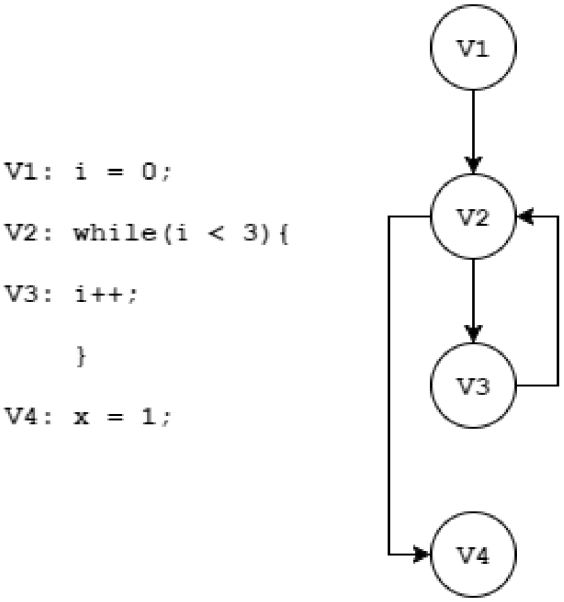
3.5 Kontrola toku programu

Ďalšou kategóriou metód, ktoré sa snažia zvýšiť spoľahlivosť programu je *kontrola toku*. Tieto metódy je možné ďalej rozdeliť na:

- **Hardvérové metódy** - zastaralá metóda, kde je využívaný *watchdog procesor* (r. 1988), rozšírenie pôvodnej myšlienky *watchdog časovača*. Vykonávaná bola kontrola toku programu, monitorovanie zbernice medzi procesorom (CPU) a pamäťou a podobne.
- **Softvérové metódy** - sú typické pre kontroly toku programu. Využívajú priraďovanie značiek každému uzlu v programovom grafe a doplnenie programu o algoritmus pre detekciu chýb. Medzi tieto techniky patria napríklad SIHFT (Software-Implemented Hardware Fault Tolerance), CFCSS (Control-flow Checking by Software Signatures) alebo ECCA (Enhanced Control-Flow Checking Using Assertions).
- **Hybridné metódy** - určité spojenie medzi softvérovými a hardvérovými metódami. Každá z nových hardvérových metód využíva softvérovú podporu aspoň čiastočne. Príkladom je CFCET - Control-flow checking using execution tracing.

Základné pojmy

- *Základný blok* - program pozostáva zo základných blokov, čo sú časti programu bez vetvení a iných inštrukcií riadenia (okrem poslednej inštrukcie). Množina všetkých základných blokov v programe sa označuje ako $V = \{v_i; i = 1, 2, 3, ..., n\}$.
- *Graf toku programu* - reprezentácia toku programu pomocou grafu P, ktorý je tvorený množinou $\{V, E\}$, kde E je množina vetvení medzi základnými blokmi [9]. Príklad ukážky grafu toku programu je vidieť na obrázku 3.8.



Obr. 3.8: Graf toku programu.

3.5.1 SIHFT - Software-Implemented Hardware Fault Tolerance

Dátová diverzita v kombinácií s časovo redundanciou môže byť základ pre softvérovo implementovanú hardvérovú odolnosť (SIHFT), s cieľom detekovať hardvérové poruchy. Táto metóda ponúka lacnejšiu alternatívu pre hardvérovú a/alebo informačnú redundanciu, ktorá je zvyčajne využívaná pre systémy s COTS procesormi - tieto často nepodporujú detekciu chýb [8].

Dátovou diverzitou sa myslí zmena originálneho programu na nový za pomoci vynásobenia všetkých konštánt v pôvodnom programe určitou hodnotou k. Napríklad, ak v pôvodnom programe je premenná `x = 3;`, v novej verzií bude `x = 6;`, teda hodnota $k = 2$. Vďaka časovej redundancie sú tieto variácie programu testované a ich výsledky porovnávané na rovnakom hardvéry, s cieľom odhaliť a detekovať možnú poruchu.

Vzhľadom na to, že Commercial off-the-shelf (COTS) sú veľmi často používané súčiastky na vesmírne technológie, metódy SIHFT teda tiež patria do tejto oblasti výskumu. Radiácia ako napríklad alfa častice alebo kozmické žiarenie môže spôsobovať prechodné poruchy v systémoch, ktoré zapríčiňujú chyby zvané *Single-Event Upsets (SEUs)*. Ako príklad SEUs je možné uviesť *bit-flip*, čo je vlastne nežiadúca zmena stavu obsahu uloženého v pamäti. To môže viesť k nesprávnym výsledkom aritmeticko-logickej jednotky (ALU). Proti takýmto účinkom sa používajú špecialne súčiastky, ktoré sú však veľmi nákladné. Ako alternatíva sa teda používajú COTS komponenty, ktoré sú lacnejšie a rýchlejšie, avšak majú málo funkcií na odolnosť voči poruchám. Tento problém sa dá riešiť práve technikami SIHFT [14].

3.5.2 ECCA - Enhanced Control-Flow Checking Using Assertions

Táto technika pre kontrolu toku využíva princíp vkladania kontrolných značiek na začiatok a koniec základných (programových) blokov. Vytvorí sa takzvaný *blokový identifikátor BID (Block IDentifier)*, ktorý musí byť unikátny a jednoznačný pre každý základný blok. Ďalej BID by malo byť prvočíslo väčšie ako 2. Takýto identifikátor sa priradí každému zo základných blokov. Okrem BID bloky obsahujú taktiež premennú $NEXT$, ktorá reprezentuje nasledujúce bloky, ktoré môžu pokračovať po danom bloku. Poslednou premennou, ktorú metóda ECCA využíva, je *id* obsahujúca BID bloku, ktorý je aktuálne vykonávaný. Jednotlivé bloky sú uzatvárané medzi dve kontrolné priradenia, vďaka ktorým je možné detekovať chyby toku programu.

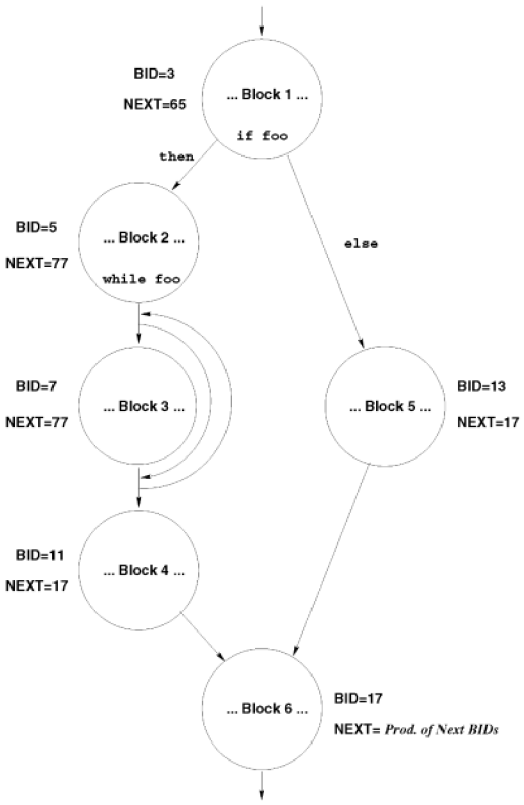
Metódu je možné implementovať nasledujúcim algoritmusom:

1. Program rozdeľ na základné bloky
2. Vytvor a priraď BID pre každý základný blok
3. Zostav graf toku programu
4. Vypočítaj a priraď celočíselnú premennú $NEXT$ (súčin všetkých BID blokov, ktoré môžu nasledovať za daným blokom)
5. Ako prvé kontrolné priradenie na začiatok bloku vypočítaj a vlož SET (rovnica výpočtu 3.1)
6. Ďalšie priradenie, $TEST$, vypočítaj a vlož na koniec bloku (rovnica výpočtu 3.2)

$$id \leftarrow \frac{BID}{\overline{\overline{(id \bmod BID)} \cdot (id \bmod 2)}} \tag{3.1}$$

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}} \tag{3.2}$$

Premenná id je nastavená na BID aktuálneho bloku vždy na jeho začiatku. Na konci bloku sa do nej uloží hodnota $NEXT$ aktuálneho bloku. V prípade ak pri priradení SET nastane delenie nulou, znamená to nesprávny tok programu. Teda ak aktuálny blok nie je nasledovníkom predchádzajúceho ($(id \bmod BID)$!= 0) alebo ak hodnota premennej id je párne číslo. Ak platí, že BID sú len prvočísla väčšie ako 2, hodnota $NEXT$ bude vždy nepárne číslo. Na konci základného bloku sa pri priradení $TEST$ nastavuje hodnota id na $NEXT$, ale ak id aktuálne vykonávajúceho bloku sa nerovná BID, $((id - BID) == 0)$, je do premennej vložené párne číslo a pri priradení SET je detekovaná chyba. Príklad implementácie metódy ECCA na grafe toku programu je znázornený na schéme 3.9 (prevzaté z [1]).



Obr. 3.9: Príklad implementácie metódy ECCA na grafe toku programu.

3.5.3 CFCSS - Control-flow Checking by Software Signatures

Rovnako ako aj predchádzajúca technika aj CFCSS je čisto softvérová metóda na kontrolu toku programu. Na veľmi podobnom princípe algoritmus priraďuje do každého základného bloku v grafe toku programu jedinečnú značku, s cieľom detekovať možnú poruchu. Tieto značky sú vypočítané a uložené ešte počas kompilácie programu.

Najskôr je program rozdelený do základných blokov tzv. uzly a podľa toku programu je zostrojený príslušný graf. Do každého uzlu je vložená značka, ktorá je reprezentovaná ľubovolným číslom (už nie je nutné dodržanie pravidla *prvočíslo väčšie ako 2*) a tiež hodnotu získanú ako rozdiel medzi značkou zdrojového a cieľového uzla. Oproti technike ECCA, ktorá určuje na konci bloku možné uzly, pre skok z aktuálneho bloku, metóda CFCSS kontroluje na začiatku aktuálneho bloku, či zo zdrojového bolo možné skočiť. Inak povedané, či je aktuálne vykonávaný základný blok programu tým správnym.

Kontrolný algoritmus pracuje s globálnou premennou G, v ktorej je uložená značka aktuálneho bloku, ďalej premenná d reprezentuje rozdiel medzi zdrojom a cieľom. Počas vykonávania programu sa pomocou operácie XOR medzi G a d, (podľa funkcie 3.3, príklad použitia 3.4) zisťuje správnosť toku programu. Ak výsledkom operácie je značka aktuálneho bloku, program pracuje správne, v opačnom prípade je detekovaná chyba. Problém by mohol nastať pri situácií, kedy je možné z viacerých zdrojov pokračovať do aktuálneho bloku. Vtedy je nutné pridať do každého z daných blokov pridať značku pre opravu D. Tento jav sa nazýva *aliasing*.

$$f \equiv f(G, d_i) = G \oplus d_i \tag{3.3}$$

$$G = G_2 = f(G_1, d_2) = G_1 \oplus d_2 \tag{3.4}$$

Výhodou techniky CFCSS je, že nevyžaduje aby operačný systém podporoval multitasking, ako je to nutné napríklad pri implementácií watchdog časovača (softvérovo). Vďaka práci s jednoduchou operáciou XOR má CFCSS oproti ECCA nižšie pamäťové aj výkonnostné nároky[9].

3.6 Klasifikácia vplyvov porúchu a chýb na chovanie jadier OS a aplikácií

Poruchy môžu spôsobovať niekoľko rôznych stavov v systéme, ktoré ho vo výsledku môžu aj nemusia ovplyvniť. Tieto stavy sa podľa [6] rozdeľujú do niekoľkých tried:

- **Bez zmeny** (angl. *Effectless*)- žiadne viditeľné dopady na funkcionalitu systému
- **Spustenie výnimky** (angl. *Exception trigger*) - program vyvolá zodpovedajúcu obslužnú rutinu (napríklad pri delení nulou alebo pri nepovolenej inštrukcie)
- **Zlyhanie systému** (angl. *System crash*) - systém prestáva fungovať
- **Zlyhanie aplikáci**e (angl. *Application failure*) - reprezentuje viacero porúch, ktoré sú ďalej klasifikované do jednotlivých kategórií podľa ich dôsledkov na beh aplikácie.

- **Nesprávny výsledok** (angl. *Incorrect output results*) - jedna alebo viaceré úlohy aplikácie poskytujú výsledky ale nie sú korektné
- **Problém riešenia v reálnom čase** (angl. *Real-time problem*) - pri operačných systémoch pracujúcich v reálnom čase može dochádzať k chybe, kedy jedna alebo viac úloh aplikácie nerešpektujú ich obmedzenia, ktoré sa týkajú práce v reálnom čase
- **Nereagujúca úloha** (angl. *Task hang*) - systém stále pracuje ale jedna alebo viaceré úlohy aplikácie nereagujú - prestali fungovať

Kapitola 4

Návrh a implementácia

Pre správny chod každého systému je veľmi dôležitá integrita dát. Ak je táto vlastnosť porušená, či už softvérovo alebo hardvérovo, je nutné aby systém túto skutočnosť zistil a vhodne zareagoval. V najlepšom prípade, ak je to možné vzhľadom na zdroje a minimálnu funkčnosť, takáto porucha by namala mať vplyv na činnosť systému.

Táto kapitola, ako je už zrejmé z jej názvu, je zameraná na návrh a implementáciu sady experimentov, ktoré ukážu vplyv porúch a chýb na chod jadra operačných systémov a spustených aplikácií nad nimi. Podstatnú časť tvorí aj teória o použitých nástrojoch a zvolených metódach. Celá kapitola je rozdelená na dve hlavné sekcie, ktoré reprezentujú zvolené typy experimentov.

Konkrétne pre experimenty boli zvolené *trvalé poruchy* zavádzané metódou *Fault injection*, teda *injektovanie porúch*. Ide o veľmi známu techniku na skúmanie a testovanie, kedy je umelo zavádzanú porucha do systému,môže ísť o hardvérovú ale aj softvérovú poruchu.

Pred samotnou implementáciou boli navrhnuté a otestované viaceré možnosti spôsobu vykonania experimentov avšak tie, ktoré boli vybrané sa ukázali ako efektívne a spoľahlivé pre potreby tejto práce.

Prvou z možností bolo využitie emulátora a virtualizéra QEMU (viz. podkapitola 4.1.2) v spojení s frameworkom pre injekciu hardvérových porúch pre platformu ARM - FIES. *Fault Injection for Evaluation of Software-based fault tolerance* skrátene FIES[1] je zjednodušene povedané rozšírenie QEMU o injekciu porúch. Podporuje simuláciu porúch, ako napríklad injekciu CPU poruchy, prístupu do pamäti a podobne. Zároveň poskytuje spätnú väzbu o diagnostikovaní chýb. Nevýhodou tohto frameworku je nekompatibilita s niektorými distribúciami (hostiteľského) operačného systému Linux, čo bolo aj dôvodom nevyužitia tejto techniky pre plánované experimenty.

Ďalšia varianta bolo použitie hardvérovej platformy FITkit 3 s využitím vývojového prostredia *The Kinetis ® Design Studio* (KDS) a s podporou *Software Development Kit* (SDK) - balíček s podpornými modulmi a funkciami. Ale s novou verziou SDK vznikli určité problémy, preto nebola zvolená ani táto technika.

Pre potreby experimentov tejto práce boli vybrané nástroje (samostatné) QEMU a Posix simulátor FreeRTOS, ktoré budú bližšie špecifikované v nasledujúcich podkapitolách.

Cieľom navrhnutých experimentov je lepšie pochopiť a klasifikovať vplyv porúch na poskytovanie služieb jadra zvolených operačných systémov a chod aplikácií, ktoré sú nad ním spustené. Vďaka tomu je možné lepšie navrhnúť mechanizmy na zvýšenie spoľahlivosti, ktoré by mali nežiadúce účinky chýb minimalizovať alebo odstrániť.

[1] https://github.com/ahoeller/fies

4.1 Injektovanie porúch do bootovacieho sektoru s využitím QEMU

V nasledujúcej podkapitole bude bližšie špecifikovaný návrh a riešenie sady experimentov, ktoré využívajú injektovanie porúch pomocou metódy *bit flipping* do bootovacieho sektoru v obraze virtuálneho stroja operačného systému Linux. Ako emulačný a virtualizačný nástroj bol vybraný QEMU.

4.1.1 Návrh experimentov

Hlavná podstata tejto sady experimentov je injektovať poruchu (zmenou bitu) v bootovacom sektore obrazu operačného systému Linux a sledovať vplyv poruchy na následný pokus o nabootovanie jadra.

Vzhľadom na to, že predmetom sledovania bolo iba bootovanie jadra Linuxu, nebolo potrebné pracovať s komplexným a zložitým obrazom virtuálneho stroja. Preto bol vybraný „malý“ a jednoduchý obraz disku[2] obsahujúci verziu 2.6.20 jadra Linuxu, X11 a potrebné funkcie na testovanie pomocou QEMU avšak bez grafického užívateľského prostredia.

Bit flipping

Jednou z možností ako injektovať poruchy do systému je využitie metódy *bit flipping*. Táto technika je jednoduchá na pochopenie a implementáciu ale zároveň aj veľmi účinná.

Bit flipping v preklade preklopenie bitu alebo zmena je veľmi častá porucha v systémoch, či už cielená alebo nie. Cielenou sa myslia napríklad kybernetické útoky, ktoré využívajú práve túto techniku. Príkladom môže byť zmena oficiálnej domény na podvodnú, ktorú útočník vytvoril použitím bit-flip na jeden bit (napr. www.fit.vutbr.cz na www.fIt.vutbr.cz), na získanie dôležitých dát.

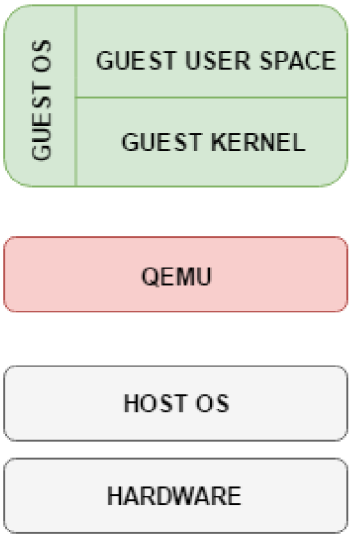
Bit flip je teda zmena bitu $0 \rightarrow 1$ alebo $1 \rightarrow 0$. Je možné tvrdiť, že zmena len jedného bitu môže v dôsledku vyvolať chybový stav systému, záleží podľa úrovne spoľahlivosti daného systému. Táto hardvérová porucha môže byť spôsobená taktiež zmenou napätia, kozmickým žiarením alebo nárastom teploty.

4.1.2 QEMU - Quick EMUlator

Quick Emulator, skrátene *QEMU* je emulátor a virtualizačný stroj, ktorý umožňuje spustenie celého operačného systému len ako ďalšiu úlohu na pracovnej ploche (schéma architektúry 4.1). *QEMU* je taktiež open-source program využívajúci dynamický preklad na dosiahnutie čo najlepšej emulačnej rýchlosti. Súčasťou jeho vlastností je aj vytváranie obrazov virtuálnych strojov, rovnako ako viac známejšie programy - VirtualBox[3] alebo VMware Workstation[4].

Využitie *QEMU* ako emulátora - riešený softvérovo, ktorý umožňuje spustiť programy s odlišnou architektúrou (napr. ARM) ako je na hostiteľskou počítači (napr. x86_64). Ale ak je architektúra virtuálneho stroja zhodná s tou hostiteľskou je možné využiť modul *KVM (Kernel-based Virtual Machine)* pre lepší výkon. V prípade použitia ako virtualizér dosahuje *QEMU* takmer natívny výkon vďaka spúšťaniu programov priamo na hostiteľskom CPU.

[2] http://wiki.qemu.org/Testing/System_Images
[3] https://www.virtualbox.org/
[4] http://www.vmware.com/products/workstation.html



Obr. 4.1: Schéma architektúry QEMU.

Je niekoľko formátov obrazov virtuálnych strojov, ktoré *QEMU* podporuje. Niekoré z nich:

- Raw images (`.img`) - binárna kópia disku (viac v nasledujúcej kapitole)
- CD/DVD images (`.iso`) - zahŕňajú obsah optického disku podľa sektorov
- QEMU copy-on-write (`.qcow2, .qed, .qcow, .cow`) - formát súboru pre obraz disku využívaný *QEMU*, veľmi flexibilný
- VirtualBox Virtual Disk Image (`.vdi`) - formát, ktorý využíva VirtualBox
- VMware Virtual Machine Disk (`.vmdk`) - formát, ktorý využíva VMware Workstation

Nástroj *QEMU* vie byť veľmi užitočný pri skúšaní odlišných operačných systémov, testovaní a experimentovaní alebo pri spúšťaní aplikácií s inou architektúrou ako je na hostiteľskom počítači [22].

4.1.3 Obraz operačného systému a jeho rozloženie

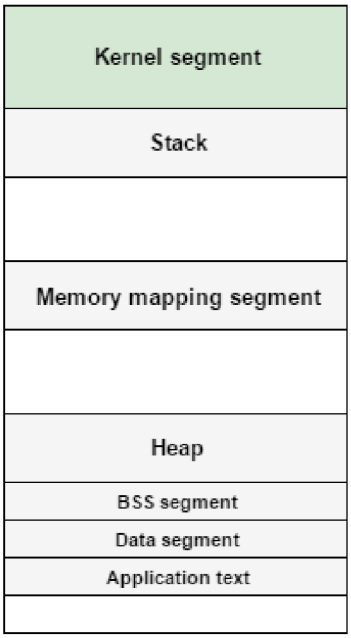
Ako už bolo uvedené v návrhu tejto sady experimentov, bol vybraný jednoduchý obraz operačného systému Linux (angl. *image*) s formátom podporujúcim QEMU - raw (`.img`).

Tento binárny formát súboru uchováva obraz pevných alebo optických diskov. Vďaka tomu, že obraz pozostáva z binárnej kópie zdrojového média podľa sektorov, výsledný súbor bude závisieť od súborového systému disku, z ktorého je obraz vytvorený (napríklad podľa verzie FAT[5]). Tieto obrazy diskov obsahujú nie len dáta z každého sektoru ale aj kontrolné hlavičky a polia s korekciou prípadných chýb [24]. Avšak formát súboru pre QEMU obsahuje iné časti (viz. 4.3).

Výhodou tohto formátu obrazu je jednoduchosť a prenositeľnosť medzi rôznymi emulátormi, nielen QEMU. V prípade, že súborový systém podporuje riedke indexovanie, výsledný súbor bude mať menšiu veľkosť ako virtuálny disk, ak je disk sčasti prázdny.

[5]File Allocation Table

Vzhľadom na to, že injektovanie porúch, v tomto experimente, bolo navrhnuté do bootovacieho segmentu, bolo nutné zistiť rozloženie jednotlivých častí v binárnej kópií disku. Keďže načítanie jadra operačného systému je prvou a hlavnou operáciou pri spúšťaní virtuálneho stroja (rovnako ako aj pri spúšťaní OS fyzicky uloženého v počítači), logicky sa bootovací sektor nachádza v prvej časti celého obrazu. Čo bolo aj otestované niekoľkými pokusmi pred spustením sady experimentov. Približné rozloženie sektorov binárneho súboru je znázornené na nasledujúcej schéme [21]:



Obr. 4.2: Schéma rozloženia segmentov v obraze virtuálneho stroja Linux.

4.1.4 Popis implementácie

Pri implementácií experimentov injektovanie porúch do bootovacieho sektoru virtuálneho stroja Linux, sa postupovalo podľa návrhu uvedeného v podkapitole 4.1.1. Pred samotným začatím bolo vykonaných pár skúšobných pokusov pre efektívne riadenie celej sady experimentov. Vzhľadom na to, že na určenie správnosti experimentu sa podieľalo viacero faktorov, ako napríklad správny informačný výpis do príkazového riadku QEMU pri pokuse o nabootovaní, možnosť nekonečného cyklu a podobne, bol zvolený manuálny systém. Tým sa myslím, že nebol vytvorený žiadny program, ktorý by automaticky riadil simuláciu a vyhodnocoval tak testy.

Postup implementácie by sa dal rozdeliť do niekoľkých krokov a to:

1. Výber náhodnej pozície v bootovacom sektore obraze virtuálneho stroja
2. Injektovanie poruchy pomocou metódy bit flipping
3. Pokus o nabootovanie jadra použitím virtualizéra QEMU
4. Vyhodnotenie
5. Obnova neporušeného obrazu virtuálneho stroja Linux

4.2 Injektovanie porúch s využitím Posix simulátora FreeRTOS

Ako už bolo vysvetlené v úvode tejto kapitoly, nasledujúca sekcia bude venovaná druhej sade experimentov a to *Injektovanie porúch s využitím Posix simulátora*. Rovnako ako v predchádzajúcej podkapitole aj tu, bude najskôr vysvetlený základný princíp návrhu pokusov a tiež, použité nástroje, techniky a metódy. V kapitole číslo 5 budú zhodnotené výsledky experimentov, podľa ktorých bude navrhnutý spôsob zvýšenia spoľahlivosti systému. Predmetom skúmania bude tak, úroveň náchylnosti systému na injektované poruchy.

4.2.1 Návrh experimentov

Návrh druhej sady experimentov je zameraný najmä na klasifikáciu vplyvu porúch na chod aplikácií spustených nad jadrom. Kvôli tomu bolo potrebné zvoliť komplexnejšie prostredie ako jednoduchý obraz virtuálneho stroja, použitý v prvej sade experimentov. Po zvážení niektorých možností (uvedených v úvode kapitoly 4), bol vybraná *Posix simulátor FreeRTOS*, ktorému bude venovaná nasledujúca podkapitola.

Aby boli výsledky experimentov čo najrozmanitejšie je vybraných niekeľko typov aplikácií, s rôznymi modifikáciami, s ktorými sa pracuje počas testovania. Čo sa vlastne dá chápať ako určitá podoba N-násobného programovania (Softvérová redundancia 3.3.2), keďže jeden princíp je implementovaný viacerými metódami. Základ tvoril program, ktorý tromi rôznymi riadiacimi algoritmami zoradil pole náhodných čísiel. Najmä pri tomto type aplikácie záleží na presnosti operácií a systému, nad ktorým pracuje. Aby bolo jadro systému, čo najviac zapojené do riadenia, vybralo sa pár druhov programov, ktoré využívajú úlohy na riadenie toku programu, ich podrobnejšie popísanie bude v podkapitole 5.2, vrámci vyhodnotenia experimentov.

Princípom tejto sady experimentov je externe injektovať trvalú poruchu metódou podobnou bit flipping, do spustiteľného súboru. Tento *Executable and Linkable Format*, skrátene *elf* súbor vznikne po kompilácií zdrojových súborov vrámci simulátora. Aj v tomto prípade je užitočné poznať rozloženie súboru, do ktorého sa bude zanášať porucha, pre efektívnejšie výsledky. Takto upravený súbor, bude spustený a jeho chod a výstup, teda výstup aplikácie, bude analyzovaný, či sa daná porucha prejavila alebo nie.

4.2.2 Posix simulátor FreeRTOS

Prostredie Posix simulátoru FreeRTOS[6] , ktorého autorom je *William Davy*, bolo vybrané na tieto experimenty, pre jednoduchú manipuláciu s ním a efektívne výsledky. Výhodou taktiež je, že nie je potrebné používať ďalšie nástroje alebo techniky ako napríklad QEMU alebo dodatočné frameworky.

Autor tento nástroj považuje za port, ktorú umožňuje systému FreeRTOS pracovať ako plánovač pre vlákna a to vrámci procesu „hostiteľského“ operačného systému. Je navrhnutý tak aby umožňoval vývoj a testovanie programov v POSIXovom prostredí. Považuje sa za simulátor pretože nevykonáva operácie v reálnom čase ale zachováva rovnaký determinizmus v riadení úloh (vysvetlené v podkapitole **FreeRTOS**). Na jeho chod tak nie je vyžadované pridanie ďalších hardvérových komponentov ako napríklad vlastné CPU, disky, pamäť RAM a podobne. Simulátor a aplikácie, ktoré nad ním bežia je možné spustiť podobne ako pri spúšťaní iných programov s posixovým štandardom a to:

[6] http://www.freertos.org/FreeRTOS-simulator-for-Linux.html

```
$ make

$ ./FreeRTOS_Posix
```

`FreeRTOS_Posix` je spustitelný súbor, ktorý vznikne po kompilácií zdrojákov vrámci simulátora. A práve do tohto súboru je zavádzaná porucha.

POSIX

Portable Operating System Interface, skrátene POSIX je IEEE[7] štandard, ktorého cieľom je dosiahnuť, čo najväčšiu kompatibilitu medzi systémami, ktoré ho využívajú, nezávisle na hardvérovej platforme. Obsahuje definície pre programove a vývojové prostredie akým je API[8] a zároveň užívateľské prostredie - príkazový riadok alebo rôzne nástroje. Tento štandard prevažne používajú Unix-ové systémy [26].

FreeRTOS

FreeRTOS je operačný systém reálneho času (angl. *Real Time Operating System* - RTOS) s modulárnym jadrom. Je na vrhnutý tak, aby poskytoval preemptivný (často nazývaný deterministický) spôsob vykonávania. Real-time vykonávanie znamená, že systém musí reagovať na určitú udalosť v striktne definovanom čase (termíne). Záruku na splnenie požiadaviek v reálnom čase je možné vykonať len vtedy, ak je možné predvídať správanie plánovača operačného systému (a preto deterministické). v FreeRTOS je zabezpečený determinizmus tým, že užívateľovi je poskytnutá možnosť pridania priorít na každé vlákno, v FreeRTOS nazývané *úloha*. Systém vďaka tomu vyhodnotí tok programu [30].

4.2.3 Radiace algoritmy

V niektorých experimentoch sa využívajú radiace algoritmy na demonštrovanie vplyvu porúch na chod aplikácie, pri ktorých sú nutné presné výpočty a operácie. Táto časť práce je venovaná stručnému úvodu k problematike radiacich algoritmov a krátkemu popisu použitých typov.

Radiace algoritmy sa využívajú na usporiadanie určitej dátovej štruktúry do špecifickej sekvencie. Táto dátová štruktúra by mala byť homogénna, teda jej položky by mali byť rovnakého typu. Najbežnejším radením je podľa numerickej veľkosti čísel ale taktiež dáta zoradiť v abecednom poradí [5].

Bubble sort

Bublinové radenie (angl. *Bubble sort*) je veľmi známy a implementačne jednoduchý radiaci algoritmus. Funguje na na princípe výberu (presúvanie max a min do výstupnej postupnosti). Algoritmus prechádza sekvenciu opakovane, pričom sa porovnávajú vždy susedné položky a v prípade obráteného usporiadania nastane výmena položiek. Nevýhodou je, že metóda má kvadratickú časovú zložitosť. Implementácia bola prevzatá a upravená z [23].

[7] Institute of Electrical and Electronics Engineers
[8] Application Programming Interface

Shell sort

Shell sort je radiaci algoritmus, ktorý je známy svojím znižujúcim sa prírastkom. Pracuje na princípe opakovaného prechádzania dátovej štruktúry a vkladania vybraných položiek. Na začiatku sa určí tzv. **krok**, ktorého veľkosť určuje vzdialenosť dvoch porovnávajúcich sa znakov. Tieto znaky sa následne porovnajú, a prípade, že ten vzdialenejší má menšiu ordinárnu hodnotu, znaky sa vymenia. Po vyhodnotení a prípadnom presune sa do operácie porovnávania dostane nasledujúci znak s ďalším o hodnotu kroku vzdialeným. Pri prvom prechádzaní je krok rovný polovici počtu znakov v danej dátovej štruktúre, pri každom ďalšom prechode sa krok zmenší o polovicu, až kým nie je rovný 1 (vrátane). Algoritmus bol prevzatý a upravený z [27].

Quick sort

Rýchle radenie (angl. *Quick sort*) je algoritmus využívajúci radenie rozdeľovaním a patrí medzi najrýchlejšie metódy zoradenia polí. Princíp práce quick sort-u je rozdelenie poľa hodnôt do dvoch častí, pričom naľavo sú položky menšie alebo rovné určitej hodnote a napravo sa nachádzajú položky väčšie. Zoradená dátové štruktúra je vtedy, keď je algoritmus postupne uplatní na všetky takto rozdelené časti a ich počet prvkov je väčší ako 2. Hodnota, s ktorou sa ostatné prvky porovnávajú sa nazýva *pivot* a môže byť zvolený buď ako medián hodnôt, náhodne alebo ako prvý prvok poľa. Algoritmus bol prevzatý a upravený z [28].

4.2.4 Executable and Linkable Format a rozloženie súboru

Táto sekcia je venovaná *Executable and Linkable Format* (ELF) súboru, ktorý bol využívaný pre potreby experimentov. Do vybraných častí ELF súboru boli injektované poruchy.

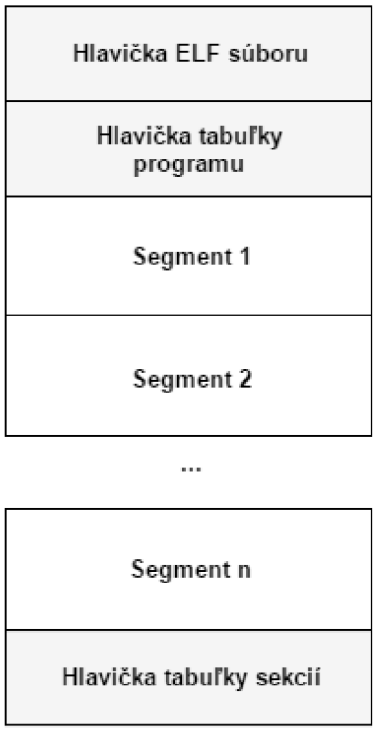
ELF je bežný formát spustiteľného súboru, ktorý je binárnou reprezentáciou programu a je určený na spúšťanie priamo na procesore. Taktiež sa využíva na uloženie linkovaných objektov, zdielaných knižníc a ladiacich výpisov. Tento formát využívajú viaceré operačné systémy ako Linux, FreeBSD, FreeRTOS a iné, pôvodne bol vytvorený pre operačný systém UNIX. Štruktúra a rozloženie ELF súboru je naznačená na nasledujúcom obrázku [11].

Každý ELF súbor sa skladá z viacerých častí ale hlavné rozdelenie je **ELF hlavička** a **súborové dáta**, ktoré za hlavičkou bezprostredne nasledujú.

ELF hlavička sa nachádza vždy na začiatku a obsahuje základné informácie o súbore ako aj niekoľko štruktúr, ktoré popisujú napríklad umiestnenie ďalších hlavičiek a celú organizáciu súboru, veľkosť hlavičky a iné.

Hlavička tabuľky programu je vlastne pole štruktúr, ktorá každá popisuje jednotlivé segmenty v súbore a informuje systém ako vytvoriť obraz procesu. Taktiež táto časť obsahuje typy segmentov, ich veľkosť alebo adresu, na ktorej sa nachádzajú. Program využíva túto tabuľku práve pri vytváraní a obrazu procesu a spúšťaní programu.

Vzhľadom na to, že každý segment je ďalej rozdelený do jednotlivých segmentov, informácie o nich ako aj ich rozloženie obsahuje posledná časť súboru - **hlavička tabuľky sekcií**. Každá sekcia má položku v tabuľke, ktorá informuje o jej názve, veľkosti a podobne. Tieto sekcie sa využívajú pri linkovaní a realokácií [29].



Obr. 4.3: Schéma rozloženia segmentov v ELF súbore.

4.2.5 Popis implementácie

Podľa návrhu v podkapitole 4.2.1 boli vykonané sady experimentov využívajúce Posixový simulátor FreeRTOS. V prvom rade bolo nutné implementovať sadu aplikácií, ktoré sa budú podieľať na testovaní. Tými boli aplikácia na radenie poľa náhodných čísel tromi rôznymi radiacimi algoritmami bez a s využitím riadenia pomocou úloh a jednoduché aplikácie pracujúce s úlohami ale s rôznymi metódami (úprava predpripravených aplikácií v balíčku na testovanie simulátora).

Vzhľadom na to, že priestor (plocha) na možné injektovanie poruchy je oveľa väčšia, ako v prípade experimentov s bootovaním jadra (4.1), bolo potrebné vykonať omnoho viac testov a preto bolo aj zvolené automatizované testovanie. Taktiež v tomto prípade je možný postup experimentovania popísať do nasledujúcich krokov:

1. Pomocou sekvenice príkazov `make clear` a `make` bol odstránený pôvodný a vytvorený nový spustitelný súbor `FreeRTOS_Posix`
2. Injektovanie poruchy do spustitelného súboru `FreeRTOS_Posix` zmenou znaku na náhodnej pozícií
3. Spustenie `FreeRTOS_Posix`
4. Porovnanie aktuálneho výstupu s originálnym (bez zanesenej poruchy), poprípade kontrola, či sa program nedostal do nekonečného cyklu

Ako už bolo uvedené vyššie experimenty boli vykonávané automatizovanie a to pomocou dvoch skriptov, ktoré je možné nájsť v priloženom dátovom médiu (CD). Jeden bol

tzv. hlavný, v jazyku Bash, ktorý riadil celé testovanie. Obsahoval cyklus, ktorý určoval počet experimentov, ktoré sa majú vykonať a taktiež pokrýval body 1., 3. a 4. Po úspešnom vykonaní bodu číslo 1. bol spustený druhý skript, v Python-e, ktorý vykonával zavádzanie poruchy do súboru. Najskôr bolo vygenerované náhodné číslo `random_num` v rozsahu, ktorá odpovedala ploche zvolenej pre injektovanie poruchy, avšak bolo nutné skontrolovať, či vygenerované číslo `random_num` je unikátne v rámci danej sady experimentov. Tento problém bol vyriešený ukladaním si predchádzajúcich vygenerovaných čísel do súboru. Na pozícií `random_num` bolo následne zmenený znak ako externe zanášaná porucha.

Kapitola 5

Výsledky experimentov a návrh mechanizmov na zvýšenie spoľahlivosti

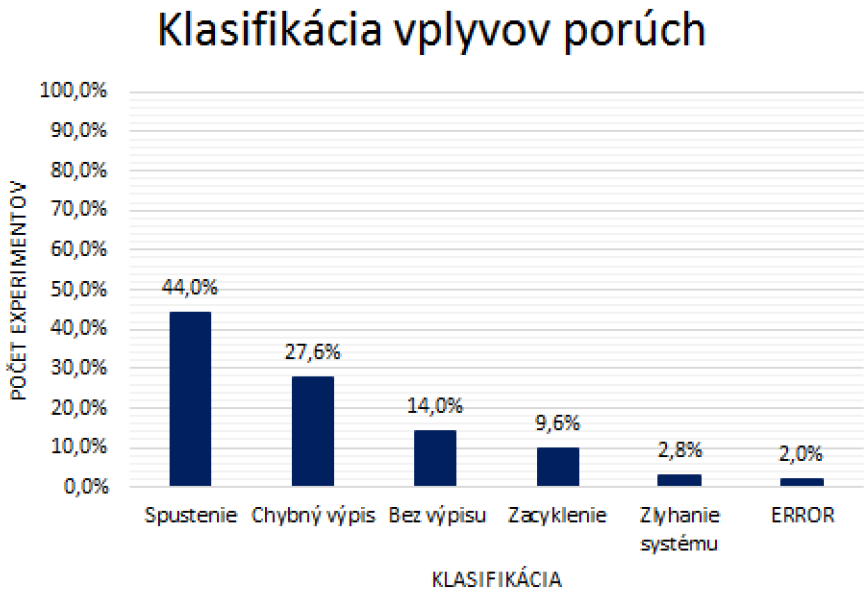
Výsledky experimentov, ktoré boli navrhnuté a ich implementácia popísaná v predchádzajúcej kapitole, sú zhodnotené v tejto časti práce. Okrem toho, sa tu čitateľ môže dozvedieť návrh mechanizmov na potlačenie vplyvov porúch a chýb systému. Podobne aj táto kapitola je členená do niekoľkých logických celkov a to vyhodnotenie experimentov injektovania porúch do bootovacieho sektoru s využitím QEMU, ďalej injektovanie porúch s využitím Posix simulátora a návrh zvýšenia spoľahlivosti. V rámci prvých dvoch sekcií je grafické znázornenie získaných a spracovaných výsledkov experimentov. Pre väčšiu prehľadnosť bola väčšina grafov presunutá do sekcie Príloha.

5.1 Injektovanie porúch do bootovacieho sektoru s využitím QEMU

Táto časť práce je venovaná vyhodnoteniu získaných výsledkov z vykonaných experimentov injektovania porúch do obrazu virtuálneho stroja Linux, ktorých návrch a implementácia bola popísaná v podkapitolách 4.1.1 a 4.1.4. Celkovo bolo spustených 250 experimentov, vzhľadom na to, že bootovacia časť obrazu nie je, v porovnaní s celým súborom, až taká veľká. Podľa grafického zobrazenia 5.1, je možné vidieť, že výsledky, teda vplyvy porúch, sa dajú klasifikovať do niekoľkých typov a to skrátene *Spustenie, Chybný výpis, Bez výpisu, Zacyklenie, Zlyhanie systému* a *ERROR*.

Pod *Spustením* sa myslí nabootovanie jadra Linuxu bez prejavenia poruchy. Takýchto prípadov bola skoro polovica z celkového počtu vykonaných experimentov. V ostatných prípadoch je možná podrobnejšia klasifikácia. *Chybný výpis* znamená, že sa do spúšťacieho okna virtuálneho stroja zobrazila iba určitá sekvencia znakov zo začiatku informačného výpisu zavádzania alebo symboly, čísla a podobne. Naopak *Bez výpisu* označuje stav kedy sa nepodarilo zaviesť jadro ale ani nebol zobrazený žiadny textový výstup. Je zjavné, že *Zacyklenie* popisuje stav, kedy sa do spúšťacieho okna vypisovala určitá sekvencia znakov v nekonečnom cykle, vtedy musel byť proces násilne zrušený. Pri *Zlyhaní systému*, ktorý nastal iba v $2,8\%$ prípadov sa okno virtuálneho stroja vôbec nezobrazilo alebo po veľmi krátkej dobe sa samo zrušilo. Posledným typom je *ERROR*, ktorý nastal v najmenšom počte prípadov, sa myslí vypísanie chybovej hlášky do okna programu.

Vďaka tomuto experimentu sa podarilo vytvoriť približnú klasifikáciu vplyvov porúch na priebeh zavádzania jadra operačného systému. Z grafu 5.1 je vidieť, že vo viac ako polovici experimentoch sa injektovaná porucha prejavila. Teda aj zmena jediného bitu v obraze má veľký dopad na systém.



Obr. 5.1: Graf zobrazujúci výsledky experimentov so zavádzaním jadra Linuxu.

5.2 Injektovanie porúch s využitím Posix simulátora FreeRTOS

Nasledujúca podkapitola je o niečo komplexnejšia a podrobnejšie členená ako predchádzajúca. Rozdelenie na jednotlivé sekcie je podľa typov vykonaných experimentov, ktoré boli navrhnuté v podkapitole 4.2.1. Všetky sekcie budú obsahovať grafické spracovanie s percentuálnym zobrazením výsledkov experimentov, ktoré by mali čitateľovi uľahčiť prácu s textom a lepšie sa zorientovať v klasifikácií. Pre získanie výsledkov bolo pre každý typ vykonaných 1000 experimentov ak text neuvádza inak.

5.2.1 Spracovanie úloh

Dôležitou vlastnosťou a súčasťou jadra operačného systému je **plánovač**. Ten má za úlohu riadiť tok programu a prideľovať každému procesu operačný čas. A práve na toto boli zamerané nasledujúce experimenty, ktoré využívali funkciu plánovača na riadenie a spracovávanie úloh rôznymi spôsobmi. Všetky budú popísané a ich princíp fungovania bližšie uvedený v nasledujúcich častiach.

Ak legenda v grafoch neuvádza inak, význam kategórií v jednotlivých grafoch bude vždy totožný. Konkrétne *Žiadny vplyv* sú prípady, kedy sa na chod aplikácie neprejavila alebo nebola spozorovaná porucha. V ostatných typoch sa porucha prejavila ako pri *Bez výpisu*,

kedy výstupný súbor neobsahoval žiadne dáta, ktoré by mal (pomocné hlášky informujúce o toku programu a jeho stave). Typ *Chybný výpis* obsahuje varianty experimentov, kedy síce výstupný súbor obsahoval dáta ale neboli správne. Podobne ako v podkapitole 5.1 *Zacyklenie* označuje stav, kedy sa program dostal do nekonečného cyklu, kým nebol zrušený.

Riadenie úloh na základe priority

Princípom aplikácie (súbor `main_003.c`), ako je možné vidieť na uvedenej časti kódu nižšie, je vytvoriť dve instancie rovnakej implementácie úlohy, ale s rozličnými parametrami. Každá instancia má svoj vlastný reťazec znakov, ktorý sa vypíše na výstup a inú prioritu. Po inicializovaní plánovača je vždy spustená úloha s vyššou prioritou. Keďže v tejto aplikácií nie je riešené predávanie riadenia medzi úlohami, úloha s nižšou prioritou nebude vykonaná. Graf A.1 popisuje klasifikáciu porúch na chod aplikácie.

```
int main( void )
{
  /* Create one of the two tasks. */
  xTaskCreate(vTask1,   /* Pointer to the function,implements the task.*/
              "Task 1", /* Text name for the task only for debbuging. */
              1000,     /* Stack depth. */
              (void*)pcTextForTask1,   /* Pass the text to be printed. */
              1,        /* This task will run at priority 1. */
              NULL );   /* We are not using the task handle. */

  /* ... and the second task at priority 2.
  The priority is the second to last parameter.
  We are creating two instances of a single task implementation.*/
  xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);

  /* Start the scheduler so our tasks start executing. */
  vTaskStartScheduler();

  return 0;
}
```

Riadenie úloh na základe zvyšovania a znižovania priorít

Podobne ako v predchádzajúcom prípade aj v tejto aplikácií (súbor s názvom `main_008.c`) sa pracuje s prioritami úloh. Rovnako sú vytvorené dve instancie jednej implementácie úloh, každá s inou prioritou, avšak je pridaný parameter na riadenie, vďaka ktorému je možné danej úlohe meniť úroveň priority.

```
/* Create the first task at priority 2.  This time the task
parameter is not used and is set to NULL.  The task handle is
also not used so likewise is also set to NULL. */
xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
```

```
/* Create the second task at priority 1-which is lower than the priority
given to Task1.  Again the task parameter is not used so is set
to NULL - BUT this time we want to obtain a handle to the task
so pass in the address of the xTask2Handle variable. */
xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );
```

Vždy sa vykoná úloha s vyššou prioritou, čo je zabezpečené tým, že keď je úloha s vyššou prioritou dokončená zníži, resp. zvýši, prioritu úlohe, ktorá obsahuje parameter na riadenie (`Task 2`) tak, aby mala nižšiu, resp. vyššiu prioritu. Výsledky experimentov sú zobrazené na grafe A.2.

```
/* Setting the Task2 priority above the Task1 priority will cause
Task2 to immediately start running (as then Task2 will have the higher
priority of the two created tasks). */
printf( "About to raise the Task2 priority\r\n" );
vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );
/* Task1 will only run when it has a priority higher than Task2.
Therefore, for this task to reach this point Task2 must already have
executed and set its priority back down to 0. */
```

Riadenie úloh pomocou fronty

Ďalšou aplikáciou , ktorej zdrojový kód je možné nájsť v `main_010.c` využíva na riadenie úloh frontu.

```
/* The queue is created to hold a maximum of 5 long values. */
xQueue = xQueueCreate( 5, sizeof( long ) );
```

Do fronty sa postupne striedavo zapisujú dve instancie rovnakej implementácie úlohy s rovnakou prioritou. Tretia úloha, ktorá má vyššiu prioritu, vyberá z fronty úlohu, ktorá je aktuálne na rade a vykoná ju. Grafické zobrazenie výsledkov jednotlivých experimentov je na A.3.

```
/* Create the task that will read from the queue.  The task is created
with priority 2, so above the priority of the sender tasks. */
xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
```

Riadenie úloh pomocou fronty so štruktúrami

Práca tohto algoritmu, v súbore `main_011.c`, je veľmi podobná predchádzajúcej avšak parametre, ktoré sa do fronty ukladajú sú dátového typu štruktúra. Klasifikáciu z tejto sady experimentov je graficky zobrazená na A.4.

```
/* Define the structure type that will be passed on the queue. */
typedef struct {
      unsigned char ucValue;
      unsigned char ucSource;
} xData;
```

Pri porovnaní všetkých výsledkov experimentov z takto sekcie je vidieť, že práve **Riadenie úloh pomocou fronty so štruktúrami** má najnižšie percento úspešnosti. Preto bolo spustené testovanie ešte raz, teda spolu 2000 experimentov, pre podrobnejšiu klasifikáciu vplyvov, za predpokladu zanedbania kategórie *Žiadny vplyv*. Graf je označený ako A.5.

5.2.2 Radiace algoritmy

Pre väčšiu rozmanitosť experimentov sa pracovalo s viacerými typmi aplikáciami a ich modifikáciou. Za ďalší typ bola zvolená aplikácia, ktorá troma rôznymi radiacimi algoritmami zoradila pole náhodných čísel podľa ich numerickej veľkosti. Najskôr boli skúmané dopady injektovaných porúch na chod aplikácie, ktorá nevyužívala spracovanie úloh (prvá sekcia tejto podkapitoly) a následne bol do programu doplnený algoritmus s touto vlastnosťou (druhá časť podkapitoly).

Radiace algoritmy nevyužívajúce spracovanie pomocou úloh

Podľa návrhu a implementácie bola vykonaná sada experimentov, kde sa injektovala porucha zmenou jedného znaku v spustiteľnom súbore binárneho formátu (graf A.6). Vrámci testovania však bola pridaná aj varianta so zmenou 4 po sebe nasledujúcich znakov (graf A.7) a následne porovnaný rozdiel medzi týmito verziami. Avšak ako je vidieť z oboch grafov dopad nie je veľmi veľký. Konkrétne rozdiel $0,8\%$ v prípade, kedy sa porucha neprejavila. Klasifikácia pri daných sadách experimentov je rozčlenená len do 3 skupín. Význam jednotlivých kategórií v grafoch je podobný ako v podkap. 5.2.1 avšak bez stavu *Zacyklenie*. Do varianty *Chybný výpis* sú zarátaná prípady, kedy porucha zasiahla znak napríklad v informačnom výpise (teda radiace algoritmy mohli pracovať správne) alebo stavy, kedy bolo zistené chybné radenie niektorých algoritmov.

Radiace algoritmy využívajúce spracovanie pomocou úloh

Ako už je z názvu možné zistiť, táto sada experimentov bola zameraná na vloženie riadiacich algoritmov do úloh a ich riadenia. Pre potreby testov však boli použité iba 2 radiace algoritmy a to *Bubble sort* a *Shell sort*, keďže *Quick sort* bol implementovaný rekurzívne. Obe úlohy mali rovnakú prioritu a vykonali sa postupne. Na príslušnom grafe (A.8) je možné vidieť, že úspešnosť aplikácie do ktorej bola injektovaná porucha je v porovnaní s inými metódami na riadenie úloh, celkom dobrá.

Injektovanie viacerých porúch súčastne

V reálnom chode systému však môžu nastať aj viaceré poruchy naraz a nie len na jednom mieste. Aj vzhľadom na to, boli simulované prípady, kedy sa zavádzali poruchy na dve a tri náhodné miesta súčasne. Na grafoch A.9 (dve poruchy) a A.10 (tri poruchy) sú výsledky jednotlivých experimentov. Pri porovnaní je vidieť značný rozdiel medzi jednotlivými kategóriami.

Vďaka týmto typom experimentom, konkrétne s tromi poruchami, kde je chybovosť väčšia, je možná podrobnejšia klasifikácia, v prípade že bude zanedbaná kategória *Žiadny vplyv*. Výsledky sú prezentované na grafe A.11.

5.3 Zhodnotenie vykonaných experimentov

Na základe výsledkov prezentovaných v jednotlivých grafoch je možné vidieť, aký vplyv má na jednotlivé systémy takýto spôsob injektovania porúch.

V prípade experimentov so zavádzaním jadra Linuxu je možno tvrdiť, že aj elementárna porucha akou je *bit flipping* môže mať veľký dopad. Môže to byť zapríčinené aj tým, že vybraný obraz bol „jednoduchý“. Tým sa myslí, že nemusel obsahovať všetky komplexné

zabezpečenia proti prípadným poruchám ako reálny operačný systém alebo jeho obraz. Samozrejme toto je len predpoklad a tvrdenie, ktorého správnosť by sa musela dokázať. To však presahuje rámec tejto práce.

Pri experimentoch s využitím Posix simulátoru FreeRTOS je vidieť, že čím komplikovanejšia technika bola využitá pri riadení úloh tým väčší vplyv mali poruchy na celkový chod aplikácie. Príkladom môže byť využitie fronty so štruktúrami, kedy prejavenie poruchy bolo až v $8,1\%$ prípadoch, pričom pri riadení s prioritami je chybovosť $5,5\%$. Je teda dôležité aby v praxi pri výbere algoritmu, napríklad na spracovanie úloh, sa dbalo aj na možný výskyt poruchy.

Za zaujímavé zistenie je možné považovať, porovnanie dopadu štvor-znakovej zmeny za sebou s jedno-znakovou poruchou a s viacerými poruchami na rôznych miestach. To môže byť zapríčinené rozložením dát v súbore, kedy pri zasiahnutí napríklad nevyužitého miesta alebo metadát, nemá vplyv na výsledný chod aplikácie. Avšak pri injektovaní viacerých porúch na rôzne miesta je väčšia pravdepodobnosť, že bude zasiahnutá dôležitá časť.

5.4 Návrh mechanizmov na zvýšenie spoľahlivosti

Poruchy, ktoré boli vybrané pre experimenty v tejto práci, boli najmä hardvérového typu a v rámci dĺžky trvania spadali do kategórie *Trvalé poruchy* (viz. 3.1). Návrh mechanizmov na zvýšenie spoľahlivosti bude zameraný práve na tieto typy.

Proti jednobitovým zmenách aké boli simulované v prvej sade experimentoch, je veľmi účinný **Hammingov kód** (Informačná redundancia, viz. 3.3.3). Tento druh zabezpečenia, okrem toho že dokáže jednobitovú poruchu detekovať, je schopný ju aj opraviť. Je známy pre použitie priamo proti poruchám v pamäťových mediách, ako napríklad v RAM. Nevýhodou však je, že na efektívne zabezpečenie je nutné pridať niekoľko (extra) kontrolných bitov, čo zaberá viac pamäte.

Taktiež je možný určitý druh **Hardvérovej redundancie** (viz. 3.3.1), kedy by sa využilo duplikovanie pamäťového média aj s obsahom, kde je uložená napr. bootovacia časť OS. Avšak táto metóda by mohla byť použitá najmä v prípade, kedy by daný operačný systém bol fyzicky uložený v počítači a nie len ako obraz virtuálneho stroja. Ďalej by bola potrebná komponenta na detekciu poruchy, a ktorá by riadila obnovenie chybového kódu zo zálohy alebo použiť princíp **trojitej modulárnej redundancie**. Avšak nevýhoda danej techniky sú často jej náklady na redundantné komponenty. Taktiež je nutné si uvedomiť, aby pri použití TMR bol výsledok správny, je prípustná porucha len v jednom module. V prípade, že by nastala chyba vo viac ako jednom module, je nutné využiť **N-násobnú modulárnu redundanciu**.

Druhá sada experimentov naznačuje ďalší typ možnej techniky na zvýšenie spoľahlivosti systému a tým je **Softvérová redundancie**, konkrétne **N-násobné programovanie**. Kedy jedna úloha je naprogramovaná viacerými rôznymi implementáciami a hlasovač rozhodne o správnosti výsledku. Bližšie rozobrané výhody a nevýhody sú uvedené v podkapitole 3.3.2. Problémy by mohli nastať, napríklad ak by porucha zasiahla hlasovací systém. V takom prípade je možné ako prevenciu proti takejto situácií využiť duplikáciu hlasovača alebo pridať zabezpečovacie kódy (**Hammingov kód**, **parita**). Čo sa týka experimentov v tejto práci, ak by chybne pracovali dva z troch radiacich algoritmov, vtedy, podobne ako pri návrhu hardvérovej redundancie, by bolo vhodné využiť **N-násobné programovanie**.

Samozrejme, veľmi efektívne na detekciu porúch a chýb sú techniky na **Kontrolu toku programu**, bližšie špecifikované v časti 3.5, taktiež možné riešenie pri vzniku poruchy hlasovacieho systému.

Kapitola 6

Záver

Cieľom tejto práce bolo analyzovať a klasifikovať vplyvy porúch a chýb na chod jadra a aplikácií nad nimi. Za týmto účelom bolo nevyhnutné sa zoznámiť so základnou terminológiou akou sú **spoľahlivosť**, **poruchy v systémoch** a jednotlivé **metódy na zvýšenie spoľahlivosti**. Následne mohli byť navrhnuté, implementované a vyhodnotené sady experimentov uvedené v jednotlivých kapitolách.

Je nutné konštatovať, že spôsobov ako testovaním splniť dané zadanie je nespočetne veľa. Príkladom sú uvedené počiatočné návrhy pre experimenty v úvode kapitoly 4. Avšak s niektorými technológiami sa vyskytli určité problémy, kvôli ktorým nemohli byť použité. Taktiež by sa dali použiť rôzne metódy, ako napríklad kontrola toku programu alebo zabezpečenie Hammingovým kódom, ktoré by mohli spresniť výsledky klasifikácie. Vzhľadom na to, že niektoré vybrané experimenty (len samotné testovanie) trvali aj viac ako hodinu (konkrétne experimenty na zavádzanie jadra Linuxu aj niekoľko hodín), neboli aplikované vyššie uvedené zlepšenia. Ďalšími návrhmi môže byť analýza softvérových (programových) chýb - *bugs*, ako napríklad prekročenie rozsahu poľa, použitie `NULL` pointera a iné.

Literatúra

[1] Alkhalifa, Z.; Nair, V. S. S.; Krishnamurthy, N.; aj.: Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, ročník 10, č. 6, Jun 1999: s. 627–641, ISSN 1045-9219, doi:10.1109/71.774911.

[2] Avižienis, A.: Fault-Tolerant Systems. *IEEE Transactions on Computers*, december 1976: str. 1304.

[3] Bagchi, S.: *Fault-Tolerant Computer System Design: Introduction to Fault Tolerant Design*. West Lafayette: Purdue University, 2016.

[4] Dubrova, E.: *Fault tolerant design: an introduction*. Stockholm, Sweden: Kluwer Academic Publishers, 2008.

[5] Honzík, J. M.: *Algoritmy a datové struktury*. Brno: Vysoké učení technické: Fakulta informačních technologií, 17-B vydání, 09.01.2017.

[6] Ignat, N.; Nicolescu, B.; Savaria, Y.; aj.: Soft-error classification and impact analysis on real-time operating systems. In *Proceedings of the Design Automation Test in Europe Conference*, ročník 1, March 2006, ISSN 1530-1591, doi:10.1109/DATE.2006.244063.

[7] Jelínek, L.: *Jádro systému Linux*. Computer Press, a.s., první vydání, 2008, ISBN 978-80-251-2084-2.

[8] Koren, I.; Krishna, C. M.: *Fault-Tolerant Systems*. San Francisco: Morgan Kaufmann Publishers Inc., 2007, ISBN 978-0-12-088568-8.

[9] Oh, N.; Shirvani, P. P.; McCluskey, E. J.: Control-flow checking by software signatures. *IEEE Transactions on Reliability*, ročník 51, č. 1, Mar 2002: s. 111–122, ISSN 0018-9529, doi:10.1109/24.994926.

[10] Plášil, F.; Staudek, J.: *Operační systémy*. Praha: SNTL, 1992, ISBN 80-03-00269-9.

[11] Polacek, M.: *Introduction to ELF*. [Online; cit. 27.4.2017].
URL http://people.redhat.com/mpolacek/src/devconf2012.pdf

[12] Pullum, L. L.: *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc. Norwood, MA, USA, 2001, ISBN 1-58053-137-7.

[13] Přenosil, V.: *FI:PV171, Úvod do teorie spolehlivosti* . Brno: Masarykova Univerzita: Fakulta informatiky, 2006.
URL https://is.muni.cz/el/1433/podzim2005/PV171/Spolehlivost.pdf

[14] Shirvani, P. P.; Oh, N.; McCluskey, E. J.; aj.: Software-Implemented Hardware Fault Tolerance Experiments COTS in Space. *International Conference on Dependable Systems and Networks (FTCS- 30 and DCCA-8)*, January 2000.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.519.1329&rep=rep1&type=pdf

[15] Slimařík, F.: *Mechanismy zvýšení spolehlivosti vestavěných systémů pracujících v reálném čase*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2009.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=9289

[16] Tanenbaun, A. S.: *Modern operating systems*. Prentice-Hall, Inc., druhé vydání, 2001, ISBN 0-13-031358-0.

[17] Vdoleček, F.: *Spolehlivost a technická diagnostika*. Brno: Vysoké učení technické v Brně: Ústav automatizace a informatiky, 2002.
URL http://autnt.fme.vutbr.cz/lab/a1-731a/FSD.pdf

[18] Velasco, A. D.; Montrucchio, B.; Rebaudengo, M.: KITO tool: A fault injection environment in Linux kernel data structures. *Microelectronics Reliability*, máj 2016: str. 153.

[19] Vojnar, T.: *Operační systémy: IOS*. Brno: Vysoké učení technické: Fakulta informačních technologií, 2008.

[20] Wilfredo, T.: *Software Fault Tolerance: A Tutorial*. NASA Langley Technical Report Server, 2000.

[21] WWW stránka: *Anatomy of a Program in Memory*. [Online; cit. 20.4.2017].
URL
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

[22] WWW stránka: *QEMU Emulator User Documentation*. [Online; cit. 20.4.2017].
URL https://qemu.weilnetz.de/doc/qemu-doc.html

[23] WWW stánka: *C program for bubble sort*. [Online; cit. 19.4.2017].
URL http://www.programmingsimplified.com/c/source-code/c-program-bubble-sort

[24] WWW stánka: *IMG (file format)*. The Wikimedia Foundation, Inc, [Online; cit. 20.4.2017].
URL https://en.wikipedia.org/wiki/IMG_(file_format)

[25] WWW stánka: *Jádro operačního systému*. The Wikimedia Foundation, Inc, [Online; cit. 22.11.2016].
URL https://cs.wikipedia.org/wiki/Jádro_operačního_systému

[26] WWW stánka: *POSIX® 1003.1 Frequently Asked Questions (FAQ Version 1.15)*. Josey, A. (The Open Group), [Online; cit. 25.4.2017].
URL http://www.opengroup.org/austin/papers/posix_faq.html

[27] WWW stánka: *Shell Sort Program in C*. [Online; cit. 19.4.2017].
URL https://www.tutorialspoint.com/data_structures_algorithms/shell_sort_program_in_c.htm

[28] WWW stánka: *Sorting algorithms/Quicksort*. [Online; cit. 19.4.2017].
URL https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#C

[29] WWW stánka: *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. TIS Committee, verze 1.2, [Online; cit. 27.4.2017].
URL http://refspecs.linuxbase.org/elf/elf.pdf

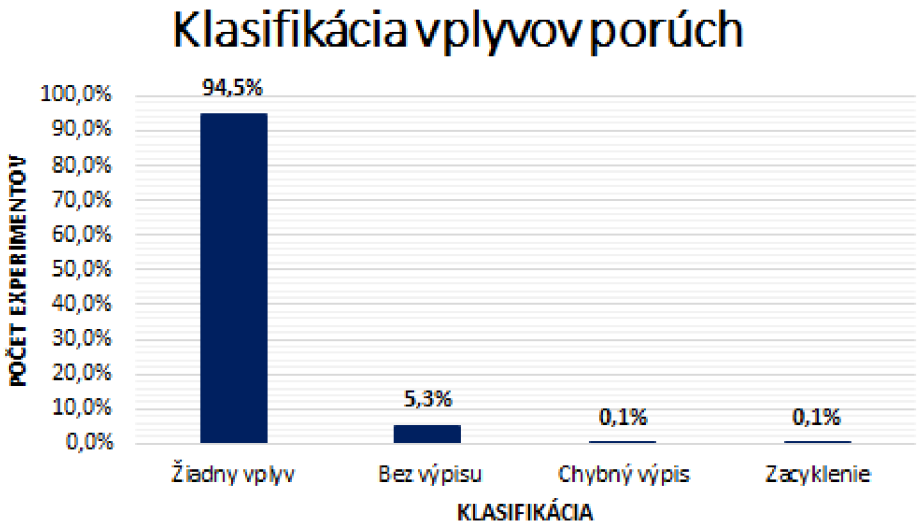
[30] WWW stánka: *What is an RTOS/FreeRTOS?* [Online; cit. 26.4.2017].
URL http://www.freertos.org/about-RTOS.html

[31] Xie, Z.; Sun, H.; Saluja, K.: *A Survey of Software Fault Tolerance Techniques*. Madison, USA, [Online; cit. 5.2.2017].
URL http://www.pld.ttu.ee/IAF0030/Paper_4.pdf

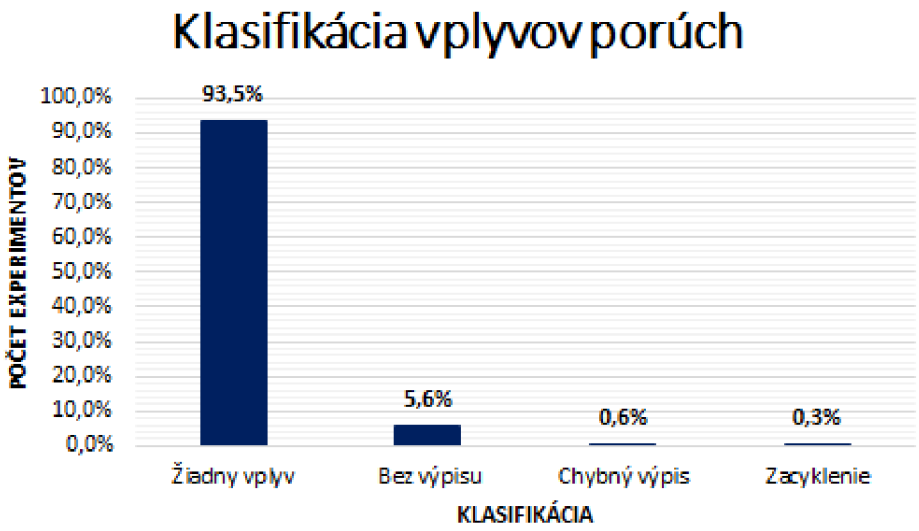
Prílohy

Príloha A

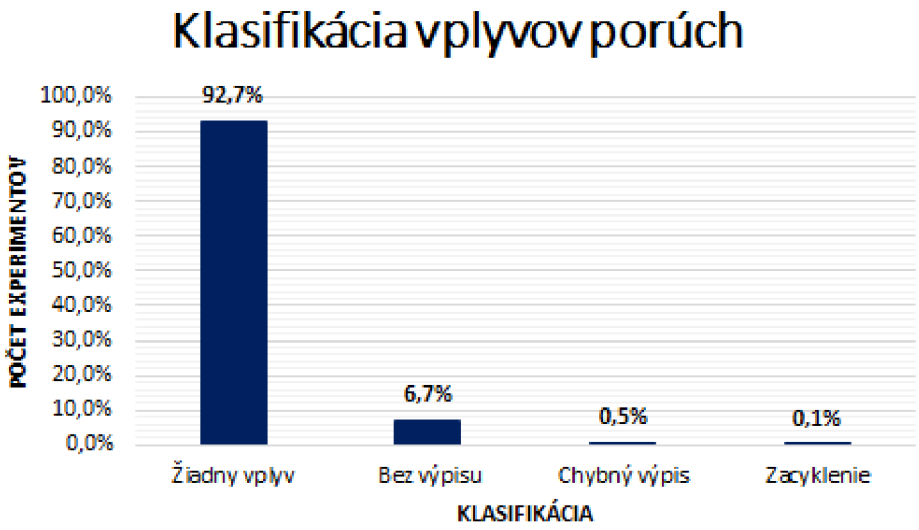
Grafické vyhodnotenie experimentov



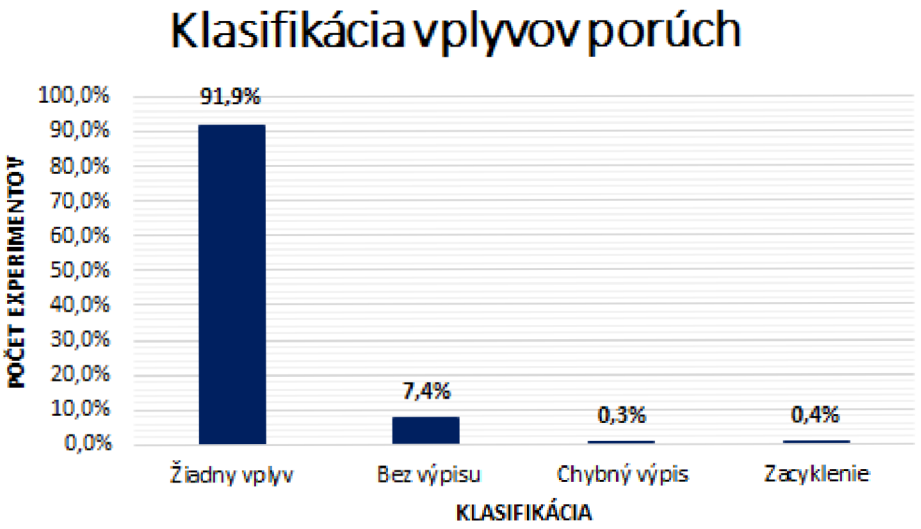
Obr. A.1: Graf výsledkov experimentov s aplikáciou na riadenie úloh na základe priority.



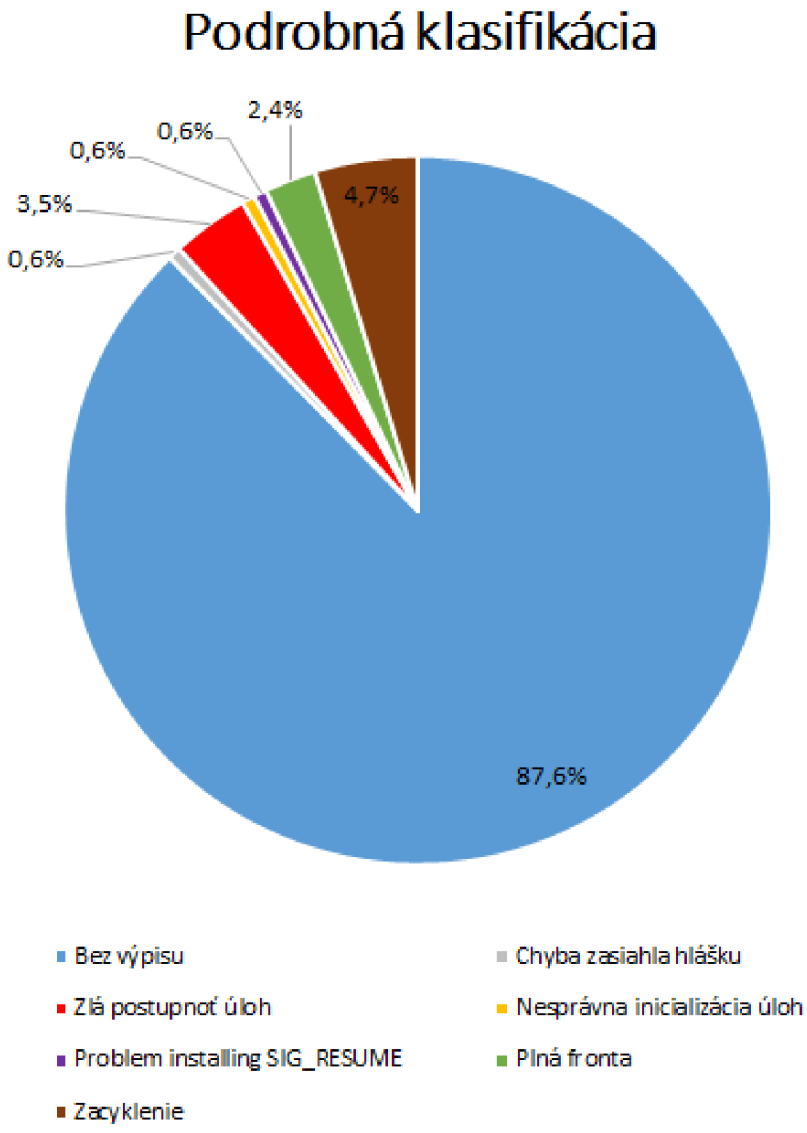
Obr. A.2: Graf výsledkov experimentov s aplikáciou na riadenie úloh na základe zvyšovania a znižovania priorít.



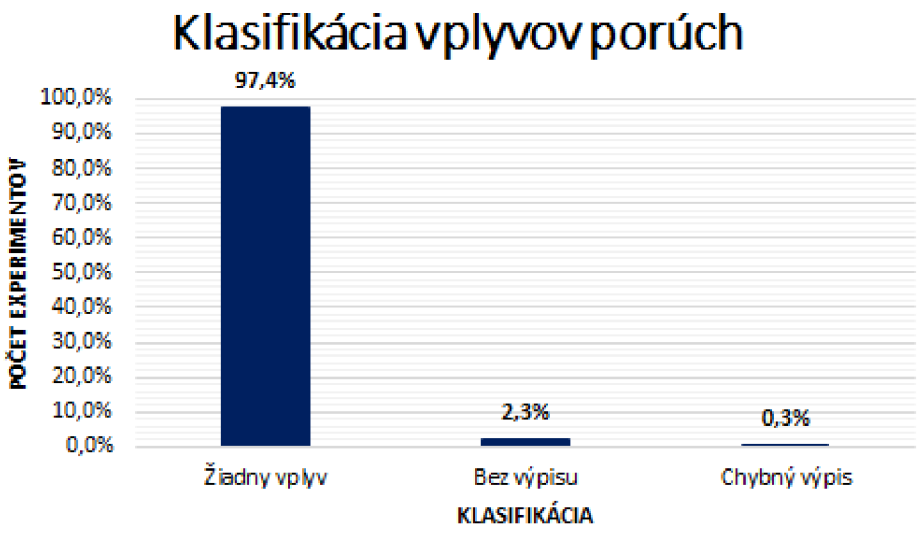
Obr. A.3: Graf výsledkov experimentov s aplikáciou na riadenie úloh pomocou fronty.



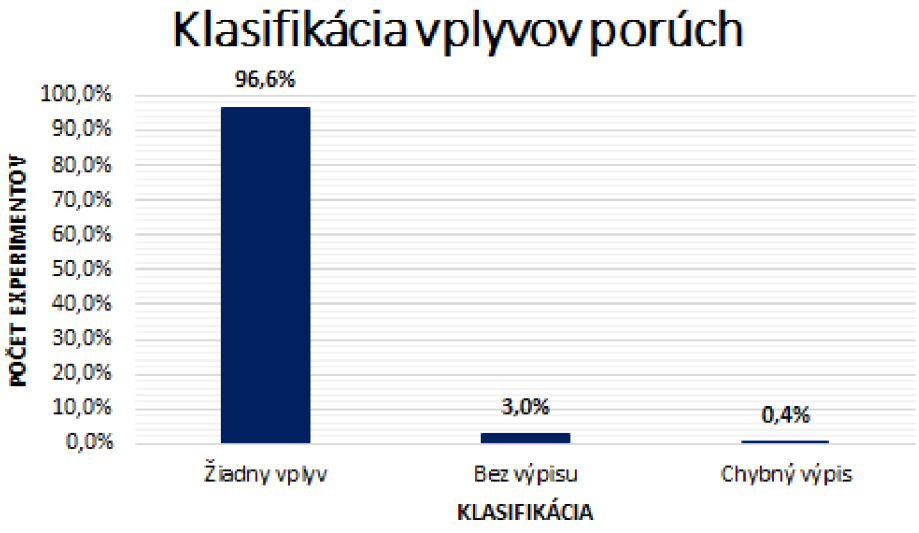
Obr. A.4: Graf výsledkov experimentov s aplikáciou na riadenie úloh pomocou fronty so štruktúrami.



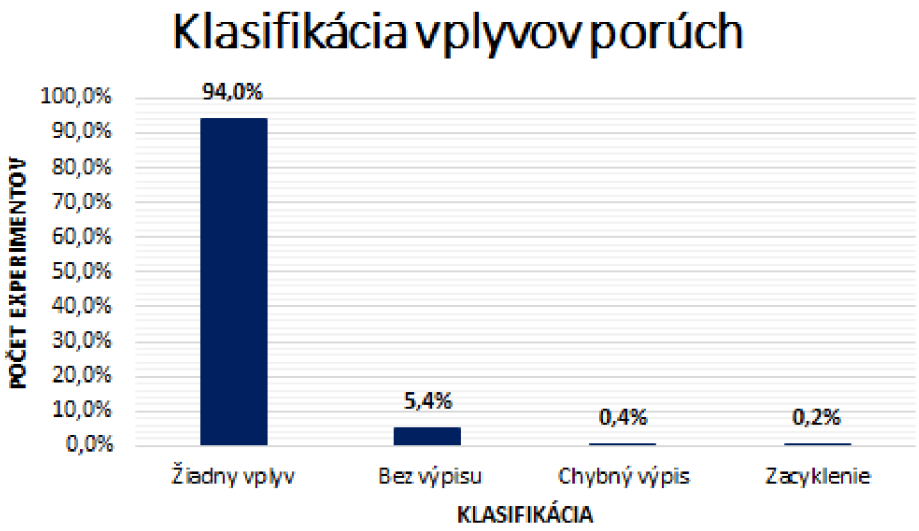
Obr. A.5: Podrobný graf výsledkov experimentov s aplikáciou na riadenie úloh pomocou fronty so štruktúrami.



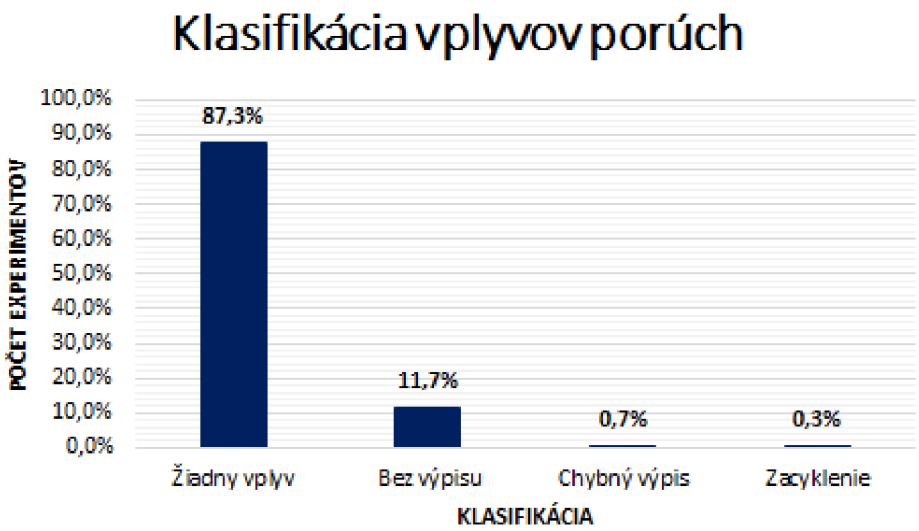
Obr. A.6: Graf výsledkov experimentov s aplikáciou na radenie poľa s jedno-znakovou poruchou.



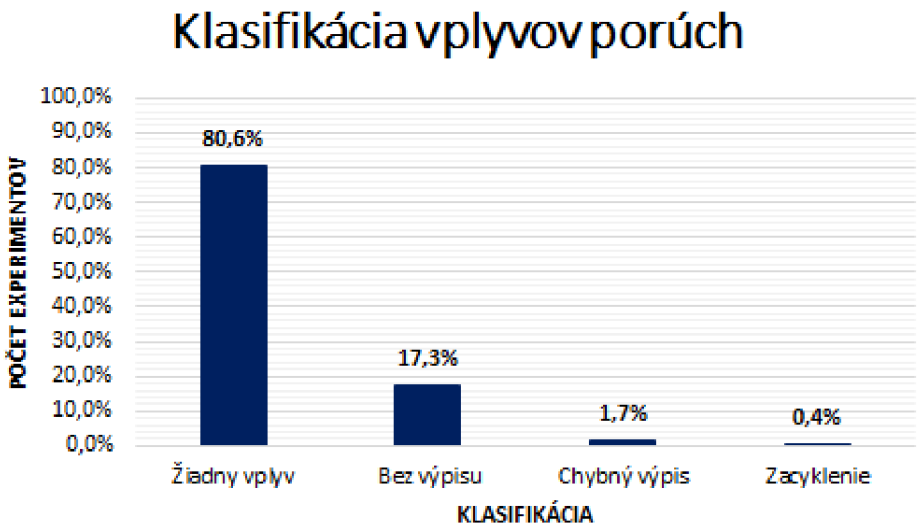
Obr. A.7: Graf výsledkov experimentov s aplikáciou na radenie poľa s štvor-znakovou poruchou.



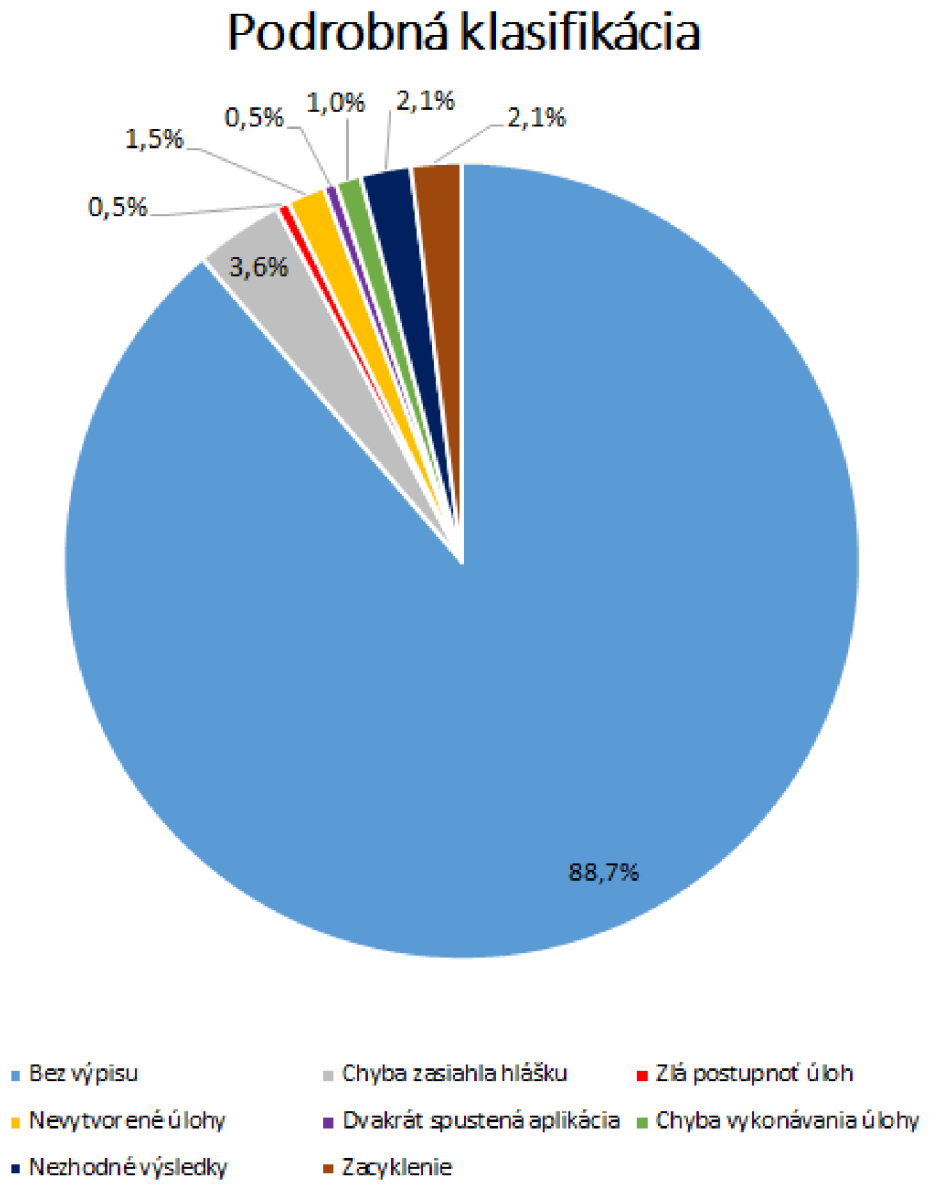
Obr. A.8: Graf výsledkov experimentov s aplikáciou na radenie pomocou úloh.



Obr. A.9: Graf výsledkov experimentov s aplikáciou na radenie úloh s dvomi injektovanými poruchami.



Obr. A.10: Graf výsledkov experimentov s aplikáciou na radenie úloh s tromi injektovanými poruchami.



Obr. A.11: Podrobný graf výsledkov experimentov s aplikáciou na radenie s tromi injektovanými poruchami.

Príloha B

Obsah priloženého pamäťového média

Priložené CD obsahuje:

- Obraz operačného systému Linux
- Posix simulátor FreeRTOS s aplikáciami
- Zdrojové súbory skriptov
- Technickú správu vo formáte pdf
- Zdrojové súbory technickej správy pre LaTeX