**Česká zemědělská univerzita v Praze**

**Provozně ekonomická fakulta**

**Katedra informačního inženýrství**


**Diplomová práce**

# Aplikace na monitorování stavu systému pro operační systém Linux

**Bc. Martin Busch**

# ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

# ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Martin Busch

Systémové inženýrství a informatika
Informatika

Název práce

**Aplikace na monitorování stavu systému pro operační systém Linux**

Název anglicky

**System monitoring application for operating system Linux**

---

**Cíle práce**

Cílem práce bude navrhnout a naprogramovat aplikaci pro operační systém Linux, která bude průběžně monitorovat stav systému. Vytvořená aplikace bude disponovat grafickým uživatelským rozhraním a bude sledovat, sbírat a zpracovávat data o využití vybraných systémových zdrojů v daném prostředí. Výstupem aplikace pak bude ucelený přehled o aktuálním stavu celého systému.

**Metodika**

Pro psaní této práce budou využívány získané poznatky z odborných knih, vybraných internetových portálů a dále také z vlastních zkušeností.

Aplikace bude napsaná v jazyce C. Pro vytvoření grafického uživatelského rozhraní bude využito sady nástrojů GTK+. Vývoj samotné aplikace bude probíhat na linuxové distribuci Linux Mint.

**Doporučený rozsah práce**

50 – 60 stran

**Klíčová slova**

C, Linux, programování, systémové informace, GUI, GTK+

---

**Doporučené zdroje informací**

JELÍNEK, L. *Jádro systému Linux : kompletní průvodce programátora.* Brno: Computer Press, 2008. ISBN 978-80-251-2084-2.

Masters, Jon Linux profesionálně : programování aplikací / Jon Masters, Richard Blum. 1. vyd.. Brno : Zoner Press, 2008. 539 s. brož. ( Encyklopedie Zoner Press ) ISBN:978-80-86815-71-8

MATHEW, N. – STONES, R. *Linux : začínáme programovat.* Praha: Computer Press, 2000. ISBN 80-7226-307-2.

Mitchell, Mark Pokročilé programování v operačním systému Linux / Mark Mitchell, Jeffery Oldham, Alex Samuel. Praha : SoftPress, 2002. 320 s. brož. ISBN:80-86497-29-1

---

**Předběžný termín obhajoby**

2019/20 LS – PEF

**Vedoucí práce**

Ing. Marek Pícka, Ph.D.

**Garantující pracoviště**

Katedra informačního inženýrství

Elektronicky schváleno dne 11. 3. 2020

**Ing. Martin Pelikán, Ph.D.**

Vedoucí katedry

Elektronicky schváleno dne 11. 3. 2020

**Ing. Martin Pelikán, Ph.D.**

Děkan

V Praze dne 29. 03. 2020

**Čestné prohlášení**

Prohlašuji, že svou diplomovou práci "Aplikace na monitorování stavu systému pro operační systém Linux" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 4.4. 2020 _____

**Poděkování**

Rád bych touto cestou poděkoval Ing. Marku Píckovi, Ph.D. za vedení mé diplomové práce. Dále bych mu rád poděkoval za pomoc a cenné rady, které mi během psaní práce poskytl.

# Aplikace na monitorování stavu systému pro operační systém Linux

**Abstrakt**

Tématem této diplomové práce je vývoj aplikací pro operační systém Linux. Pro zobrazení určitých programovacích postupů v teoretické části a i psaní samotné praktické části je použit programovací jazyk C, který úzce souvisí s historií Linuxu.

V teoretické části práce jsou uvedeny obecné postupy a principy jak lze sestavovat a ladit programy na Linuxu. Dále jsou zde uvedeny různé nástroje, které lze při vývoji programů využívat a také jak přidat do programu grafické uživatelské rozhraní.

V praktické části je navrhnuta a následně vytvořena aplikace s grafickým uživatelským rozhraním, která průběžně sbírá a zpracovává různé informace o systému. Získané informace pak zobrazuje v pro člověka čitelném formátu. Ze získaných informací pak aplikace umožňuje sestavit přehledný report, který je možné vytisknout případně uložit do souboru např. .pdf.

**Klíčová slova:** C, Linux, programování, systémové informace, GUI, GTK+

# System monitoring application for operating system Linux

**Abstract**

The topic of this thesis is the development of applications for operating system Linux. The programming language C, which is closely related to the history of Linux is used to display programming procedures in the theoretical part and to write the practical part itself.

In the theoretical part of the thesis are given general procedures and principles of how to build and debug programs on Linux. There are also listed various tools that can be used to develop programs and how to add a graphical user interface to the program.

In the practical part is designed and then created application with graphical user interface, which continuously collects and processes various information about the system. The information is then displayed in a human-readable format. The application then allows to create a report that can be printed or saved in a file such as .pdf.

**Keywords**: C, Linux, programming, system information, GUI, GTK+

# Obsah

# Seznam obrázků

# Seznam tabulek

# Úvod

Programování a linux k sobě už od vzniku linuxu neodmyslitelně patří. Zejména programování v jazyce C, ve kterém je napsáno i samotné linuxové jádro. I když linux u většiny běžných uživatelů počítače nepatří mezi nejpopulárnější operační systém, tak pro vývojáře se jedná o velmi dobrý systém právě proto, že i na vývoji a rozvoji linuxu se podílejí z velké části právě programátoři. Velkou výhodou většiny linuxových systémů oproti konkurenčním operačním systémům je, že je lze stáhnout a používat bez jakýkoliv poplatků. Dále je pak k dispozici velké množství dostupného a zajímavého softwaru, různé nástroje, které dokážou v různých případech značně ulehčit uživateli práci.

S Programy na linuxových systémech je možné se setkat a pracovat ve dvou hlavních režimech/rozhraních. Buďto mohou pracovat v textovém rozhraní příkazového řádku (CLI), ve kterém se s aplikací komunikuje skrze příkazový řádek pomocí příkazů nebo mohou mít grafické uživatelské rozhraní (GUI), které je snad pro všechny uživatele stolních počítačů dobře známo, neboť moderní operační systémy disponují grafickým režimem. Oproti práci s programy ovládané přes příkazový řádek je orientace v aplikacích s GUI pro většinu lidí podstatně jednoduší a intuitivnější. Ovšem každé z uvedených rozhraních má své výhody i nevýhody. Pomocí grafického rozhraní lze lépe vytvářet složitější aplikace, se kterými se pak i jednodušeji pracuje a které i nabízejí více možností. V případě rozhraní příkazového řádku si uživatel musí zapamatovat řadu příkazů a čím komplexnější program, tím je pravděpodobně i více příkazů. To pro běžného uživatele nemusí být optimální cesta. Na druhou stranu komunikace s programem skrze příkazy přes příkazový řádek může být v řadě případů značně efektivnější.

Tématem této diplomové práce je vývoj aplikací pro operační systém Linux. Jsou zde uvedené obecné postupy a principy, jak lze aplikace pro operační systém linux vytvářet. Následně je na základě uvedených postupů vytvořena aplikace s grafickým uživatelským rozhraním, která bude průběžně monitorovat stav sledovaného systému.

# Cíl práce a metodika

## 1.1 Cíl práce

Cílem práce bude navrhnout a naprogramovat aplikaci pro operační systém Linux, která bude průběžně monitorovat stav systému. Vytvořená aplikace bude disponovat grafickým uživatelským rozhraním a bude sledovat, sbírat a zpracovávat data o využití vybraných systémových zdrojů v daném prostředí. Výstupem aplikace pak bude ucelený přehled o aktuálním stavu celého systému.

## 1.2 Metodika

Pro psaní této práce budou využívány získané poznatky z odborných knih, vybraných internetových portálů a dále také z vlastních zkušeností.

Aplikace bude napsaná v jazyce C. Pro vytvoření grafického uživatelského rozhraní bude využito sady nástrojů GTK+. Vývoj samotné aplikace bude probíhat na linuxové distribuci Linux Mint a pro zápis zdrojového kódu bude použit standardní textový editor. Zdrojový kód bude rozdělen pro lepší přehlednost a správu do několika souborů v závislosti na jeho povaze.

# 2 Teoretická východiska

## 2.1 Operační systém Linux

Linux jako takový je z technického hlediska jen jádro, které tvoří nejnižší vrstvu operačního systému.[2] Většina lidí si ale pod tímto označením představí kompletní operační systém.[1] Na tvorbě linuxového jádra začal pracovat finský student Linus Torvalds v roce 1991 na univerzitě v Helsinkách. Od jeho jména je i slovo Linux odvozeno. Hlavní motivací bylo vytvoření bezplatného operačního systému jakým byl v té době unixový systém Minix.[7]

Operační systém Linux je vyvíjen díky spolupráci velkého počtu lidí z různých zemí po celém světě. Jedná se o svobodný operační systém, což znamená, že je zdarma a celý jeho zdrojový kód je volně dostupný. Lze ho zkoumat, upravovat, distribuovat apod. Tím se odlišuje od ostatních operačních systému, jako je Windows či macOS. Na běžných linuxových distribucí je nabízeno nespočet aplikací, které bývají převážně zdarma, ale některé jsou dostupné i za poplatek. Rozmanitost programů na Linuxu je velmi bohatá. Lze tu používat programy pro práci s grafikou, vývoj programů, práci s textem a mnoho dalších typů programů. Neustálý vývoj a růst Linuxu nemá na starosti jedna konkrétní firma, ale mnoho programátorů po celém světě. [6]

Linux jako operační systém se dnes dá charakterizovat vlastnostmi jako systém, který je bezpečný, stabilní, rychlý, úsporný, a který dbá na soukromí.[8]

### 2.1.1 Linuxové distribuce

Linux neznamená jeden konkrétní operační systém. Jedná se pouze o jádro. Výsledná linuxová distribuce pak kombinuje linuxové jádro s dalším svobodným softwarem se kterým tvoří kompletní operační systém.[9] Ve výsledku se jednotlivé distribuce více či méně mezi sebou liší, ale pořád se jedná o Linux.[10]

V konečném součtu existuje velké množství linuxových distribucí, které jsou velmi rozmanité a které se mohou lišit v mnoha aspektech např. zaměření, grafické prostředí, obsažený software a další věci. Zaměření linuxových distribucí lze členit například podle toho, zda je distribuce určena pro komerční nebo nekomerční použití, pro stolní počítače nebo servery, pro zkušené uživatele či běžné uživatele

apod. Výběr konkrétní distribuce před instalací je velice důležitý a je nutné si nejprve rozmyslet k jakému účelu bude systém sloužit. V tabulce 1 je uveden seznam několika vybraných linuxových distribucí, které patří mezi nejznámější a nejoblíbenější mezi uživateli.

| Distribuce | Popis |
|---|---|
| **Ubuntu** | Založený na distribuci Debian, obsahuje velké množství softwaru a má velmi dobrou podporu hardwaru.[11] |
| **Linux Mint** | Distribuce postavená na Ubuntu, která je velmi oblíbená a uživatelsky přívětivá.[12] |
| **Fedora** | Distribuce, která vychází z Red Hat Linuxu. Vyvíjen komunitou vývojářů, kteří mají podporu firmy Red Hat. Vhodná i pro nové uživatele.[13] |
| **Gentoo** | Jedná se o velmi flexibilní distribuci, kde si lze celý systém sestavit podle vlastních požadavků.[30] |
| **openSUSE** | Jedná se o jednu z hlavních linuxových distribucí, má dobrou podporu hardwaru. Počítač s openSUSE lze využívat jako desktop či domácí nebo firemní server.[14] |
| **Red Hat Linux** | Populární linuxová distribuce, kterou vyvíjí firma Red Hat. |
| **Debian** | Distribuce, která je plně vyvíjena komunitou.[13] |
| **Mandriva** | Moderní operační systém, který lze bez problému využívat jak na domácím či kancelářském počítač.[15] |
| **Knoppix** | Jedná se o live distribuci, která běží přímo z DVD nebo CD.[16] |
| **CentOS** | Distribuce, která je založená na Red Hat Enterprise Linux a která je dostupná zdarma.[17] |

**Tabulka 1: Seznam nejznámějších linuxových distribucí**

## 2.2 Programovací jazyk C

Programovací jazyk C se zrodil již před mnoha lety. První návrh a vývoj jazyka započal někdy na začátku sedmdesátých let 20. století v Bellových laboratořích. Jazyk vyvíjel výzkumný pracovník Denis Ritchie za pomoci Briana Kernighana. Jejich motivací bylo vytvoření programovacího jazyka, který by dokázal

řešit nízkoúrovňové problémy a který by se dal použít i pro programování operačních systémů.[5]

Jedná se o univerzální programovací jazyk, který je úzce spojen se systémem UNIX a jeho dlouhou historií. Samotný systém Unix je napsán v jazyce C, stejně tak jako velká část programů, které na něm běží.[4]

Častou situací bývá, že pokud se řekne slovo programování v kombinaci se slovem Linux, mnoho lidí představí právě programování v jazyce C. Ačkoli je v tomto jazyce napsána velká část linuxových programů a i samotné linuxové jádro, neznamená to, že jiné jazyky nejsou podporovány nebo není vhodné je používat. Programovací jazyk C je základním programovacím jazykem v linuxu, ale jinak není svázán s žádným konkrétním operačním systémem. Jak bylo ale řečeno, není jediným jazykem, pomocí kterého se dají na linuxu psát programy. Naopak je zde možnost využívat velké množství programovacích či skriptovacích jazyků.[1] Mezi pravděpodobně ty nejrozšířenější jazyky, které programátoři při vývoji na linuxu využívají jsou: C/C++, Java, Perl, Python, Javascript apod.

Na začátku osmdesátých let vzniká programovací jazyk C++, který vyvinul dánský programátor Bjarne Stroustrup. Jazyk C++ je rozšířením jazyka C, který umožňuje objektově orientovaného programování a další možnosti, které jazyk C nenabízel. Jazyk C++ byl standardizován po relativně dlouhé době od jeho vzniku a to v roce 1998.[5]

## 2.3 Kompilační nástroje

Aby bylo možné napsaný program v jazyce C spustit, je potřeba nějaký překladač neboli kompilační nástroj, který soubor se zdrojovým kódem přeloží a udělá z něj výsledný spustitelný soubor. Pro překlad existuje v linuxu kolekce kompilačních nástrojů s označením GCC neboli GNU Compiler Collection, která je dostupná na většině linuxových distribucích. Pomocí GCC je tedy možné kompilovat zdrojové kódy napsané v jazyce C, ale také podporuje i řadu jiných jazyků např. Java, C++ nebo Fortran.[2] Následujícím zápisem se spustí kompilace programu z příkazové řádky:

```
gcc [-volby] nazev_souboru.c
```

### 2.3.1 Volby kompilátoru gcc

Při kompilaci napsaného programu pomocí gcc je možné předat kompilátoru řadu různých voleb. Pomocí konkrétních voleb se kompilátoru sdělují informace, jakým způsobem má pracovat, co se má zobrazit apod. Volby, kterými lze funkce kompilátoru ovlivnit se dají rozdělit do několika kategorií podle toho jakou funkci zastávají např. volby pro ladění, pro úrovňové upozornění, jazykové volby apod. Celkový počet možných voleb je poměrně velký a jejich výčet s popisem lze nalézt v manuálových stránkách. Tabulka č.2 obsahuje přehled voleb, které bývají nejčastěji používány.

| Volba | Popis |
|---|---|
| **-o název** | nastaví se název výsledného spustitelného souboru po provedení kompilace. Pokud se tato volba nepoužije, výsledný soubor bude pojmenovaný a.out. |
| **-c** | použitím volby -c proběhne kompilace souboru se zdrojovým kódem bez linkování např. při kompilaci přemístitelného kódu přidaného následně do statické knihovny apod.[1] |
| **-g** | přeložení kódu s volbou -g přidá do výsledného souboru speciální informace pro ladění, které pak může využívat např. ladící nástroj gdb. |
| **-Wall** | pro zobrazení všech možných varování. |
| **-pedantic** | volba, která dohlíží na respektování standardu C a pro zobrazení všech možných varování o nerespektování daného standard.[2] |
| **-Wformat** | volba pro dohled správného použití funkcí printf a scanf.[2] |
| **-ansi** | volba pro kontrolu jazykového standardu jazyka C (C90). |
| **-std=** | pro nastavení kontrolovaného jazykového standardu. |
| **-l<library>** | používá se pro přidání knihovny. Pokud se daná knihovna nenachází ve standardním adresáři, je nutné cestu ke knihovně ještě specifikovat volbou -L. |

**Tabulka 2: Volby kompilátoru gcc**

### 2.3.2 Přeložení a spuštění programu

Jakmile je zdrojový kód programu napsaný a uložený v textovém souboru, přichází na řadu jeho kompilace neboli přeložení, aby šel program spustit. Kompilace spočívá v přeložení souboru se zdrojovým kódem do podoby spustitelného programu. Výsledný spustitelný program je pak určitý binární soubor, kterému už procesor rozumí.[1] Pro spuštění programu je tedy nejprve nutné předat textový soubor se zdrojovým kódem kompilátoru pro jeho přeložení:

```
gcc -o nazev_programu program.c
```

Volbou -o je nastaven název výstupního souboru na "*nazev_programu*". Pokud by se při kompilaci jasně nedefinoval název výsledného souboru pomocí volby -o, tak by se automaticky pojmenoval *a.out*. Soubor *program.c* je pak samotný textový soubor se zdrojovým kódem, který má kompilátor přeložit. Pokud se nacházíme ve stejném adresáři jako kompilátorem vytvořený soubor, stačí pak pro spuštění programu použít následující zápis:

```
./nazev_programu
```

## 2.4 Make

Pomocí programu *Make* lze usnadnit a urychlit proces kompilace zdrojových souborů. Jednoduší programy mohou být napsány pouze v jednom zdrojovém souboru. V takovém případě je přeložení zdrojového souboru poměrně jednoduchý proces. Složitější programy se už ale skládají z více zdrojových souborů, kde je nutné každý soubor přeložit a sestavit nakonec výsledný program. Program *Make* ale sám o sobě výsledné soubory vytvářet neumí. Potřebuje ještě speciální soubor s instrukcemi, který se nazývá *makefile.* V Souboru *makefile* jsou definovány závislosti souborů a pravidla, která popisují, jak vytvořit cíl (obvykle nějaký jeden soubor). Program Make tedy načte soubor makefile a na základě toho určí, jaký cílový soubor je potřeba vytvořit. K vytvoření finálního cíle je kolikrát nutné nejprve vytvořit několik dílčích cílů. O tom, jak budou uplatňována pravidla a vytvářeny cíle určí program Make právě ze souboru *makefile*.[1]

Cíl neboli cílový soubor se píše v souboru *makefile* na nový řádek a hned za něj se píše znak dvojtečky. Za dvojtečku se pak zapisují názvy souborů, na kterých

cíl závisí. Pod řádek s cílem se na nový řádek zapisují konkrétní příkazy, které ale musí být odsazeny tabulátorem. Program Make také pozná, jaké soubory byly změněny a je nutné je rekompilovat. Komentáře se do souboru *makefile* vkládají pomocí znaku #, který se přidá před komentář. Přiklad toho, jak se zapisuje do souboru makefile zobrazuje následující příklad:

```
# soubor makefile

program: program.o
      gcc -o program program.o

program.o: program.c p.h
      gcc -c program.c
```

Pro kompilaci už stačí jen spustit program Make. Pokud se soubor makefile nejmenuje makefile, je nutné pomocí volby -f specifikovat název souboru. Po spuštění programu se v konzoli vypíšou příkazy, které byly provedeny viz obrázek 1.

```
pc@pc-pc:~/program/c_program$ make -f Makefile
gcc -c program.c
gcc -o program program.o
```

**Obrázek 1: Program make**

## 2.5 Textové editory a vývojová prostředí

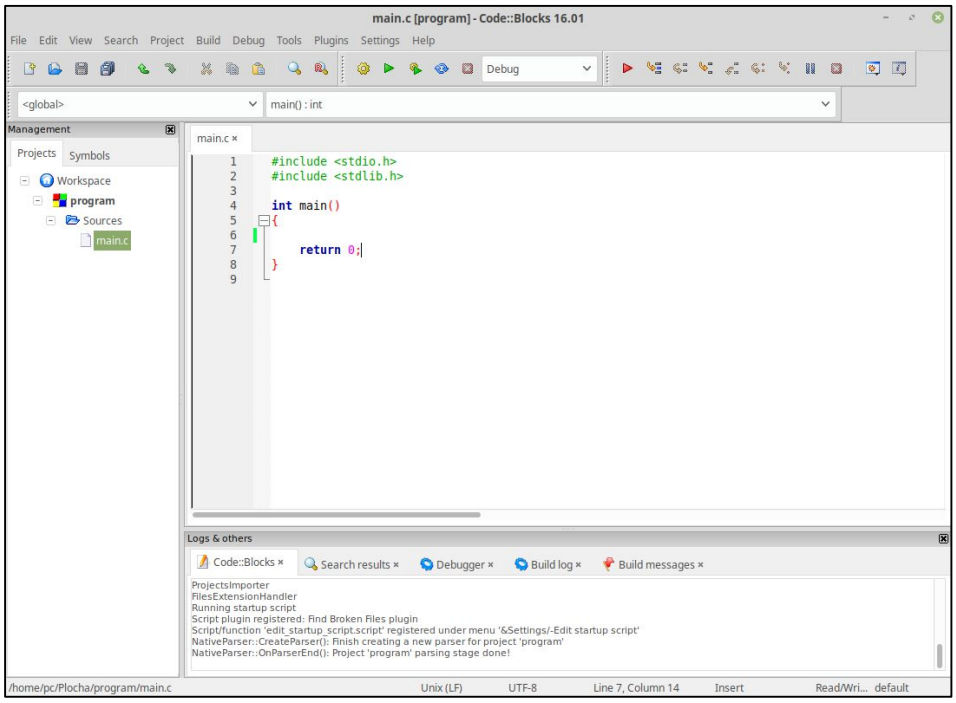
Zdrojový kód programu je potřeba někam zapisovat. Standardně se zdrojový kód programu napsaný v jazyce C zapisuje do obyčejného textového souboru s příponou .c. Pro vývoj programu lze využívat různě propracovaná vývojová prostředí, které nabízejí řadu nástrojů. Každá linuxová distribuce má v základu předinstalovaný nějaký textový editor, který úplně postačuje pro napsaní a uložení zdrojového kódu. Na linuxových distribucích s pracovním prostředím GNOME lze většinou nalézt jednoduchý textový editor s názvem *gedit*, který například umožňuje očíslování řádek či zvýrazňování syntaxe různých jazyků (C, C++, Java, Python, HTML, Perl). [18]

Mezi editory, které jsou především využívané zkušenými programátory patří textové editory Vi a jeho nástupce Vim (Vi IMproved). Tyto editory lze použít pro editování jakéhokoliv textu, ale jsou velmi vhodné pro psaní programů. Tyto editory jsou

ze začátku poměrně složité na ovládání a naučení se klávesových zkratek. Po nějaké době se ale mohou stát mocným a výkonným nástrojem programátora.

Pro vývoj programů lze také využít různá vývojová prostředí (zkráceně IDE), které navíc nabízí řadu funkcionalit. Tyto prostředí už v sobě zahrnují editor zdrojového kódu, kompilátor, ladící nástroje, kontrolu syntaxe a různé další nástroje.

Mezi jedno z nejlepších vývojových prostředí na Linuxu, které je zdarma a ve kterém lze vytvářet programy v jazyce C a C++ je prostředí s názvem Code::Blocks a které existuje také i ve verzi pro Windows.




**Obrázek 2: Vývojové prostředí Code::Blocks**

## 2.6 Používání knihoven

Pod slovem knihovna se ve světě programování rozumí nějaká kolekce či soubor předem napsaných funkcí, které poskytují různé funkcionality a které lze opakovaně použít i v jiných programech.[2] Každá knihovna obsahuje většinou soubor sobě příbuzných funkcí např. matematická knihovna, která obsahuje sadu funkcí, které se využívají pro běžné matematické operace nebo knihovna <string.h>,

která zase obsahuje funkce pro práci s řetězci. Využívání knihoven je pro programátora velmi užitečné, protože není potřeba vymýšlet vlastní složité algoritmy, když požadovanou funkcionalitu již zastupuje funkce v nějaké knihovně. Je samozřejmě zapotřebí znát jaké knihovny jsou k dispozici a jaké funkce daná knihovna obsahuje, ale toto všechno se dá najít v dostupné dokumentaci. [19]

Knihovny mohou být k dispozici ve dvou typech.[2] Prvním z těchto typů jsou knihovny označovány jako statické a druhým knihovny označovány jako sdílené. Každý z těchto dvou typů knihoven nabízí jisté výhody, ale i nevýhody.

### 2.6.1 Statické knihovny

Statická knihovna je ve své podstatě jen kolekce objektových souborů, které obsahují přemístitelný kód. Tento typ knihoven bývá také proto označován jako archivy a podle konvence má statická knihovna u názvu příponu .a. Název knihovny vždy začíná řetězcem lib.[3]

Pokud program využívá služeb nějaké statické knihovny, tak kódy požadovaných funkcí jsou přímo vloženy do výsledného spustitelného souboru.

Pomocí jednoduchého postupu je možné vytvářet a spravovat vlastní statické knihovny. K tomuto procesu se využívá program s názvem *ar*. Tento program má řadu voleb, kterými se specifikuje, co má přesně dělat:

- **p** : vytiskne konkrétní soubor z archivu na standardní výstup (print)
- **c** : vytvoření archivu (create)
- **r** : vložení souboru do archivu a pokud již v archivu existuje soubor se shodným jménem, tak jej nahradí (replacement)
- **d** : odstraní soubor z archivu (delete)
- **t** : zobrazí obsahu daného archivu (table)
- **s** : přidá nebo zaktualizuje index archivu
- **x** : extrahuje soubor z archivu (extract)

Pro vytvoření vlastní statické knihovny se musí nejprve napsat funkce, které budou skrze knihovnu dostupné. Příklad vytvoření statické knihovny, která bude

obsahovat dvě jednoduché funkce je uveden v následující ukázce. Jednotlivé funkce jsou napsané a uložené v souborech *tx1.c* a *tx2.c*

```
#include<stdio.h>
void tiskfce1()
{
   printf("Jsem funkce 1\n");
}
```

```
#include<stdio.h>
void tiskfce2()
{
   printf("Jsem funkce 2\n");
}
```

Ještě je vhodné vytvořit hlavičkový soubor s názvem např. *tx.h*, který bude obsahovat definice obou funkcí. Kompilace souborů s volbou *-c* vytvoří dva objektové soubory *tx1.o* a *tx2.o*, které obsahují zkompilovaný kód, ale nejedná se o spustitelný program.

```
gcc -c tx1.c tx2.c
```

Jsou-li připravené objektové soubory, finální statická knihovna se vytvoří zapsáním následujícího příkazu:

```
ar -cr libtext.a tx1.o tx2.o
```

Jak bylo řečeno, je nutné, aby název knihovny začínal řetězcem *lib* a měl *.a* příponu. Použitím voleb *cr* se po programu požaduje, aby se vytvořil nový archiv a přidaly se do něj dva objektové soubory. Tímto způsobem se tedy vytvoří nová statická knihovna, kterou lze již využívat v různých programech. Při kompilaci programu se na knihovnu odvolává přes volbu -l a protože je umístěna v jiném adresáři než v tom standardním, je nutné volbou -L specifikovat místo, kde se má hledat.

```
gcc -o nazevprogramu nazevprogramu.c -L. -ltext
```

Napsaným znakem tečky bezprostředně za volbou -L se říká, že se má knihovna hledat v aktuálním adresáři. Jméno knihovny stačí už zapsat zkráceně bez řetězce lib a přípony .a.[1]

Mezi nevýhody statických knihoven patří zejména to, že pokud dojde k aktualizaci knihovny např. k opravě nějaké chyby, je nutné každý program, který využívá aktualizovanou knihovnu znovu zkompilovat. Další nevýhodou je situace, kdy je napsáno více programů využívající danou statickou knihovnu. Přemístitelný kód je totiž rozkopírován do každého programu zvlášť a zabírá tedy větší diskový a paměťový prostor. Na druhou stranu jako výhoda je považováno, že programy využívající statické knihovny jsou dobře přenositelné na jiné počítače.[3]

### 2.6.2 Sdílené knihovny

Sdílené knihovny se liší od těch statických v tom, že kódy používaných funkcí ze sdílených knihoven se nevkládají přímo do výsledného programu. V programu jsou jen odkazy na sdílený kód z knihovny, který je dostupný za běhu programu.[1] Současně sdílenou knihovnu může tedy využívat (sdílet) různý počet spuštěných programů.[3] Obdobně jako u statických knihoven začíná název sdílené knihovny řetězcem lib, ale v tomto případě se používá přípona *.so*.

Použití sdílených knihoven řeší nevýhody knihoven statických. Pokud tedy dojde k úpravě či aktualizaci sdílené knihovny, není nutné rekompilovat každý program, který sdílenou knihovnu využívá. Dále pak programy využívající služeb sdílených knihoven zabírají menší diskový prostor, protože funkce z knihoven nejsou přímo vkládány do výsledného programu.[3]

Pro zjištění jaké sdílené knihovny program používá lze dosáhnout pomocí programu *ldd*, kterému se jako argument předá název programu. Následně se vypíše seznam sdílených knihoven využívaných programem.

```
ldd nazev_programu
```

```
linux-vdso.so.1 (0x00007fff9418a000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f93c48d1000)
/lib64/ld-linux-x86-64.so.2 (0x00007f93c4ec4000)
```

**Obrázek 3: Výstup z programu ldd**

Ve výstupu lze vidět, že zadaný program využívá sdílenou standardní knihovnu jazyka C *libc.so.6*. Další položky jako *ld-linux-x86-64.so* a *linux-vdso.so* se starají o načítání sdílených knihoven a určitá systémová volání.

## 2.7 Ladění programu pomocí ladících nástrojů

Protože vývojáři jsou jen lidé, tak vývoj jakéhokoliv programu bývá často doprovázen řadou různých chyb, kterých se vývojáři neúmyslně dopouští. Kvůli chybám pak program nebo jeho části nepracují tak jak bylo původně zamýšleno.

Chyby při vytváření programu mohou být různého druhu. Samotnou chybu je nutné nejprve identifikovat a následně pomocí vhodné techniky či nástroje odstranit. Chyba může vzniknout i před samotným programováním při specifikaci či návrhu programu. Pokud dojde k nesprávné specifikaci nebo návrhu programu, tak důsledkem bude, že program nebude z velkou pravděpodobností pracovat podle představ. Před samotným programováním je nutné si nejprve pořádně promyslet, co by měl daný program dělat, zda tomu rozumíme, jakou by měl mít program konstrukci apod.[1]

Z pohledu programátora bývá jako nejčastější chyba, které se programátor dopouští chyba v programování. Asi nejběžnější chyba při psaní programu je obyčejný překlep, kdy programátor neúmyslně nesprávně napíše např. název nějaké funkce ze standardní knihovny a kompilátor pak danému slovu nerozumí. Takovéto chyby se velice často odhalí právě během kompilace programu. Existují samozřejmě i jiné chyby, kterých se programátor může dopustit a které se ve většině případů během kompilace neodhalí např. pokud neoprávněně přistupujeme k nějakému místu v paměti. K odhalení chyb v programech jsou v linuxu různé ladící nástroje.[1]

Ladící nástroje jsou programy, které pomáhají odhalit různé druhy chyb a zjistit, proč program nepracuje tak jak je u něj předpokládáno. Ladících nástrojů na linuxu existuje celá řada a každý takovýto nástroj je zaměřen na řešení určitého typu problému.

### 2.7.1 GDB

Nástroj GDB neboli gnu debugger je textově orientovaný nástroj, který slouží k debugování.[1] Jedná se o velmi populární a užitečný nástroj pomocí kterého se vyhledávají a odstraňují chyby v programu.[20]

Tento nástroj umožňuje procházet kód programu krok po kroku a hledat příčinu problému.[3] Postupně lze tak například zjistit jakých hodnot proměnné během chodu programu nabývají, jak se vyhodnocují určité podmínky apod.

Aby bylo možné získat a zobrazit požadované informace, je nutné při kompilaci programu přidat volbu *-g*. Tímto se docílí toho, že kompilátor přidá ke spustitelnému souboru speciální informace, které následně použije nástroj GDB k tomu, abychom mohli program procházet a ladit dle potřeby.[20]

Pro představu jak lze nalézt v program místo, kde nastala nějaká chyba s pomocí nástroje gdb je níže uveden příklad jednoduchého programu, který převádí text na velká písmena. Zdrojový kód příkladu s názvem *znaky.c*:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>

int main()
{
      char *znaky = "text"
      char z;

      printf("%s \n, znaky");

      for(int i = 0; i < strlen(znaky); i++)
      {
           z = znaky[i];
           znaky[i] = toupper(z);
      }
      printf("%s \n, znaky");
      return 0;
}
```

Program se zkompiluje standardním způsobem spolu s přidanou volbou *-g* pro možnost ladění:

```
gcc -g -o znaky znaky.c
```

Pokud by se program spustil klasickým způsobem: *./znaky*, zobrazí se hláška o neoprávněném přístupu do paměti. Pro nalezení místa v programu, na kterém došlo k této chybě pomůže právě nástroj gdb, kterému se předá jako argument název program, který chceme procházet a ladit.

```
gdb ./znaky
```

Po spuštění se v terminálu zobrazí nejprve nějaké základní informace o verzi, licenci apod. Nakonec je zde uveden řetězec *(gdb)* za který se píšou příkazy, které se mají provést. Je zde i dostupná nápověda k jednotlivým příkazům, která se zobrazí zadáním slova *help*. Samotný program se spustí příkazem *run*.

```
(gdb) run
Starting program: /home/pc/program/c_program/znaky
text

Program received signal SIGSEGV, Segmentation fault.
0x000055555555473e in main () at znaky.c:46
46                  znaky[i] = toupper(c);
(gdb) where
#0  0x000055555555473e in main () at znaky.c:46
(gdb)
```

**Obrázek 4: Nástroj gdb**

Z výstupu je patrné, že k chybě dochází na řádku 46 ve funkci *main*. Je to z toho důvodu, že zápisem *char *znaky ="text"* se vytvoří znakový ukazatel, který ukazuje na řetězcový literál, který je ale pouze ke čtení a nelze tedy měnit. Aby program fungoval správně, lze udělat drobná úprava v podobě zápisu *char znaky[] = "text"*, kterým se nahradí původní zápis *char *znaky ="text"*. Takto se vytvoří pole znaků, kde už je možné jednotlivé položky v poli měnit.

Užitečnou pomůckou při debuggováni jsou tzv. breakpointy (body přerušení). Pomocí breakpointů se docílí toho efektu, že se běh programu zastaví v místě, ve kterém je breakpoint umístěn a z tohoto místa lze program procházet po jednotlivých krocích, prohlížet obsah proměnných apod. Bod přerušení se umístí pomocí příkazu *break*, takže například *break main* umístí breakpoint ihned na první řádek funkce main nebo příkaz *break 5* ho zase umístí na pátý řádek v programu apod. Po zastavení program v místě breakpointu se mohou pomocí příkazů *next* a *step* provádět postupné kroky. Příkazem *print* lze prohlížet obsahy konkrétních

proměnných. Příkazem *list* se zobrazí část zdrojového kódu. Pro pokračování běhu programu lze použít příkaz *cont*.[3]

### 2.7.2 Valgrind

Valgrind je volně dostupný open source software pod licencí GNU General Public License verze 2, který spadá do řady nástrojů, pomocí kterých lze ladit programy.[21] Valgrind obsahuje následujících šest nástrojů:

- **Memcheck**: používá se pro detekci různých problémů, které souvisejí se správou pamětí např.: čtení a zápis do již uvolněné paměti, úniky paměti, nesprávné uvolnění paměti, používáni neinicializované paměti apod.
- **Cachegrind**: profiler cache paměti, provádí simulaci cache paměti v procesoru
- **Callgrind**: rozšíření Cachegrindu
- **Massif**: profiler haldy
- **DRD**: detekuje problémy ve vícevláknových C a C++ programech
- **Helgrind**: detekuje chyby synchronizace v C a C++ programech, které využívají pthreads.[23][22]

Valgrind je tedy velmi mocný a užitečný nástroj pomocí kterého lze hledat v programech příčiny mnoha problémů. Jak lze Valgrind použít je zobrazeno v následujícím příkladu. Nejprve je uveden kód programu s názvem *cisla.c*, který dynamicky alokuje paměť pro pole 20ti celých čísel:

```
#include<stdlib.h>
int main()
{
      int *cisla;
      cisla = (int *)malloc(20*sizeof(int));

      for(int i = 0; i < 20; i++)
      {
           cisla[i] = i;
      }

      return 0;
}
```

Program je nutné nejprve zkompilovat a poté se může spustit Valgrind pro kontrolu, zda nejsou nějaké problémy např. s únikem paměti.

```
valgring --leak-check=yes ./cisla
```

```
==2420== HEAP SUMMARY:
==2420==     in use at exit: 80 bytes in 1 blocks
==2420==   total heap usage: 1 allocs, 0 frees, 80 bytes allocated
==2420==
==2420== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2420==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2420==    by 0x10865B: main (cisla.c:7)
==2420==
==2420== LEAK SUMMARY:
==2420==    definitely lost: 80 bytes in 1 blocks
==2420==    indirectly lost: 0 bytes in 0 blocks
==2420==      possibly lost: 0 bytes in 0 blocks
==2420==    still reachable: 0 bytes in 0 blocks
==2420==         suppressed: 0 bytes in 0 blocks
==2420==
==2420== For counts of detected and suppressed errors, rerun with: -v
==2420== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

**Obrázek 5: Výstup z programu Valgrind**

Z výstupu, který je zobrazen na obrázku 5, lze vidět, že po skončení programu nebylo uvolněno 80 bytů, které byly dynamicky alokovány pomocí funkce *malloc* na řádku 7. Počet neuvolněných bytů z výpisu přesně odpovídá počtu bytů, který byl alokován funkcí *malloc* na řádku 7. Zde byla alokována paměť pro 20 celých čísel typu *int*, kde celočíselný datový typ integer má 4 byty.

Aby se předešlo tomuto problému, je nutné při ukončování programu uvolnit námi alokovanou paměť. Stačí tedy přidat na konec programu řádek *free (cisla);* a takto dojde k uvolnění alokované paměti.

## 2.8 Vytváření GUI s knihovnou GTK+

Pod pojmem počítačový program si většina běžných lidí představí nějaké "okno" na obrazovce počítače, kde jsou rozmístěny různé ovládací a zobrazovací prvky se kterými uživatel interaguje. Grafické uživatelské rozhraní umožňuje uživateli ovládat aplikaci jednoduchým způsobem za pomoci klávesnice a myši. Pro asi většinou běžných uživatelů je obtížně představitelné, kdyby se aplikace ovládaly jen skrze příkazové řádky pomocí speciálních příkazů, kterým daný program rozumí.

Obohacením aplikace o grafické rozhraní se uživateli značně usnadní ovládání a práce s programem.

V linuxu lze využít několik různých knihoven pro přidání grafického uživatelského rozhraní do aplikace. Mezi nejrozšířenější knihovny umožňující vytvořit grafické rozhraní na linuxu patří GTK+ a Qt. Na knihovně GTK+ je postavené desktopové prostředí GNOME a na knihovně Qt zase prostředí KDE.[1]

### 2.8.1 O knihovně GTK+

GTK neboli GIMP toolkit je multiplatformní sada nástrojů, která umožňuje vytvoření grafického uživatelského rozhraní (GUI). Samotné GTK je napsáno v jazyce C, ale je uzpůsobeno tak, aby kromě jazyka C podporovalo i řadu dalších programovacích jazyků např. C++, Python, Java, Pascal, Perl, Ruby, Javarscript a další. Záleží také o jakou verzi GTK se jedná a některé jazyky jsou podporovány jen částečně např. Fortran.[24]

V případě GTK se jedná o svobodný software, který je součástí GNU Projektu. Licenční podmínky pro GTK - GNU LGPL umožňují použití všemi vývojáři bez nutnosti platit jakékoliv poplatky.[25]

GTK+ je ve své podstatě knihovna, která obsahuje řadu různých předpřipravených ovládacích prvků zvaných widgety. Tyto prvky pak ulehčují programátorům jejich práci při vytváření aplikací s grafickým uživatelským rozhraním.[1]

GTK+ závisí na následujících knihovnách:

- **Glib -** Glib poskytuje mnoho užitečných datových typů, maker, metody pro práci s řetězci nebo soubory.[26] Původně byla součástí knihovny GTK+, ale byla osamostatněna, aby ji bylo možné využívat i v programech bez grafického uživatelského rozhraní[2]
- **GObject -** GObject poskytuje objektově orientovaný systém s jednoduchou dědičností v C.[2]
- **Cairo -** Cairo je 2D vektorová grafická knihovna umožňující vytvářet kvalitní grafiku, která je nezávislá na výstupním zařízení.[2]
- **GDK -** knihovna GDK je využívána pro vykreslování grafických prvků

- **Pango** - knihovna Pango je využívána pro vykreslování textu v mnoha mezinárodních jazycích.[2]
- **GIO** - knihovna, která poskytuje virtuální souborový systém, který dovoluje vstupně-výstupní operace s místními nebo vzdálenými soubory na jednom místě.[26]
- **ATK** - knihovna pro rozhraní zajišťující přístupnost.[27]

### 2.8.2 Okno aplikace

Každá aplikace disponující grafickým uživatelským rozhraním má tzv. hlavní okno, ve kterém jsou umístěné jednotlivé ovládací a zobrazovací prvky. Tyto prvky poskytují nějakou funkčnost, skrze ně se aplikace ovládá a zobrazují se uživateli informace.

Při tvorbě grafického rozhraní je tedy nejprve nutné vytvořit právě prázdné okno aplikace, do kterého se pak budou vkládat a rozmisťovat zmíněné grafické prvky (widgety). Následujícím zápisem kódu dojde k vytvoření prázdného okna:

```
#include <gtk/gtk.h>
int main (int argc, char *argv[])
{
      GtkWidget *window;
      gtk_init(&argc, &argv);
      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
      gtk_window_set_position(GTK_WINDOW(window),
                              GTK_WIN_POS_CENTER);
      gtk_window_set_default_size(GTK_WINDOW(window),
                                  500, 250);
      g_signal_connect(window,"destroy",
                       gtk_main_quit, NULL);
      gtk_widget_show(window);
      gtk_main();
      return 0;
}
```

V první řadě je potřeba zahrnout do zdrojového souboru pomocí direktivy include knihovní hlavičku GTK+. Před voláním jakékoliv funkce GTK+ se musí nejprve pomocí funkce *gtk_init*, které se předají argumenty příkazového řádku argc a argv, inicializovat knihovny GTK+.

Následuje volání funkce *gtk_window_new*, které se předá hodnota říkající jakého typu má být vytvořené okno. V tomto příkladě je uvedena hodnota GTK_WINDOW_TOPLEVEL, která říká, že má být vytvořené běžné okno s rámečkem. Funkcemi *gtk_window_set_position* a *gtk_window_set_default_size* se nastaví pozice umístění okna na obrazovce a defaultní velikost okna po spuštění aplikace. Dále následuje volání funkce pro vykreslení samotného okna *gtk_widget_show(window)*, které je předán odkaz na nové okno.[1] Poslední funkce *gtk_main()* spouští hlavní smyčku a běží dokud není zavolána funkce *gtk_main_quit()*.

Nyní je potřeba napsaný program přeložit a spustit. Kód je uložen v souboru s názvem *okno.c* a jeho přeložení se provede pomocí kompilátoru gcc:

```
gcc `pkg-config --cflags gtk+-3.0` okno.c -o okno `pkg-
    config --libs gtk+-3.0`
```

Po přeložení a spuštění programu se na obrazovce monitoru zobrazí prázdné okno aplikace viz obrázek 6.

```
./okno
```


**Obrázek 6: GTK+ okno aplikace**

### 2.8.3 Widgety

Každá aplikace s grafickým uživatelským rozhraním má ve svém okně umístěné určité grafické prvky prostřednictvím kterých se s aplikací pracuje. V GTK+ se těmto prvkům říká widgety. I samotné okno aplikace je také widget. V knihovně GTK+ lze nalézt řadu užitečných připravených ovládacích prvků, se kterými se lze v aplikacích s grafickým uživatelským rozhraním běžně setkat. Zde je výčet těch nejzákladnějších ovládacích prvků:

**GtkButton**

GtkButton je widget představující klasické tlačítku u kterého se obvykle po kliknutí na něj provede nějaká akce. Ovládací prvek v podobě tlačítka se vytvoří pomocí funkce *gtk_button_new* nebo *gtk_button_new_with_label*, která má následující předpis:

```
GtkWidget * gtk_button_new_with_label(const gchar *str);
```

Od GtkButton je dále odvozeno několik dalších typů tlačítek, které díky svým možnostem bývají také nepostradatelnou součástí aplikací například:

- **GtkCheckButton** - představuje klasické zaškrtávací políčko (check-box)
- **GtkRadioButton** - pro vytvoření skupiny zaškrtávacích tlačítek, které se použije v případě, kdy je nutné vybrat právě jednu volbu z několika možných.
- **GtkToogleButton** - tlačítko, které vzhledově vypadá jako klasické tlačítko. Má dva stavy a nabízí v podstatě stejnou funkčnost jako GtkCheckButton.
- **GtkLinkButton** - odkaz, který je vázán na konkrétní URL adresu.

**GtkLabel**

Bez popisků různých částí aplikace nebo jednotlivých ovládacích prvků by byla navigace v aplikaci značně nepřehledná. K přidání jednoduchého popisku, kterým se zobrazuje nějaká textová zpráva se v GTK+ používá widget GtkLabel. Nový popisek se vytvoří pomocí funkce *gtk_label_new* a jako parametr se ji předá textový řetězec, který má widget zobrazovat. Funkce má následující předpis:

```
GtkWidget * gtk_label_new(const gchar *str);
```

**GtkEntry**

GtkEntry představuje vstupní jednořádkové pole pro zadání textového řetězce. Z tohoto vstupu lze tedy číst a dále zpracovávat zadaný text. Dále je možné nastavovat, jaké znaky mohou být do textového pole zapsány, jejich počet apod.[1] Vstupní pole pro zadání jednořádkového textového řetězce se vytvoří pomocí funkce *gtk_entery_new*, která má následující předpis:

```
GtkWidget * gtk_entry_new(void);
```

Pro každý Gtk widget je pak definována řada specifických funkcí, pomocí kterých lze s daným ovládacím prvkem provádět různé operace, nastavovat jeho vlastnosti, chování apod.

Knihovna GTK+ poskytuje i řadu dalších widgetů, které pokrývají mnoho jiných funkcionalit a s pomocí kterých lze vytvářet komplexnější grafické uživatelské rozhraní. Mezi takové widgety patří například:

- **GtkMenuBar** - používá se k vytvoření nabídky/menu aplikace.
- **GtkListBox** - vertikální kontejner používaný pro vytvoření seznamu. Jednotlivé řádky seznamu mohou být dynamicky řazeny, filtrovány, přidávány.[28]
- **GtkNotebook** - používá se v případě, kdy je zapotřebí rozdělit okno aplikace na několik stránek/záložek, mezi kterými se lze přepínat a zobrazovat tak na každé stránce jiný obsah.
- **GtkTreeView** - ovládací prvek, který umožní vytvořit seznam a stromově zobrazovat data ve formátu podobném tabulce. Lze tak vytvářet různé pohledy na data.[1]

### 2.8.4 Rozmísťování prvků v okně aplikace

Aby byla aplikace dobře ovladatelná, je nutné také vhodně rozmístit veškeré prvky v okně aplikace. Jednotlivé prvky aplikace není vhodné umisťovat pomocí

pevně daných souřadnic, protože na různých typech zařízení by se nemusely zobrazovat dle očekávání. Lepší volbou je použití relativního způsobu rozmisťování prvků. V GTK+ se pro rozmísťování ovládacích prvků používají kontejnery. Základním kontejnerem je GtkBox, do kterého se vkládají různé ovládací prvky, ale i další kontejnery obsahující také ovládací prvky. GtkBox podle nastavení jeho orientace uspořádává vložené widgety do řádku nebo sloupce. Nový box/kontejner se vytvoří pomocí funkce *gtk_box_new*:

```
GtkWidget * gtk_box_new(GtkOrientation orientation,
                        gint spacing);
```

Prvním parametrem se specifikuje jestli má mít box vertikální nebo horizontální orientaci. Druhým parametrem se pak nastaví jak velká má být mezera mezi prvky uvnitř boxu v pixelech.

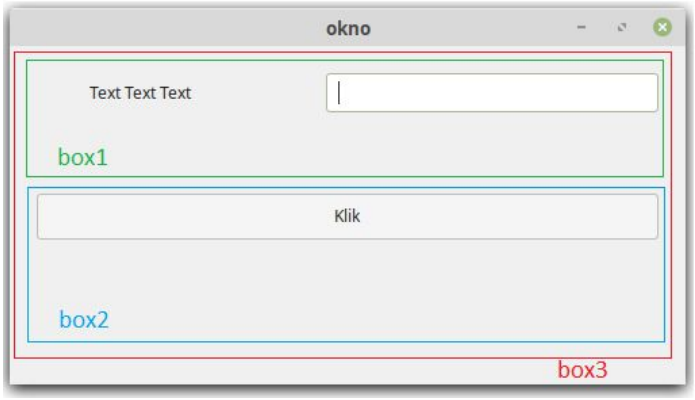
Konkrétní ovládací prvek se do boxu vloží přes funkci *gtk_box_pack_start* nebo *gtk_box_pack_end*. Rozdíl v těchto funkcí je pouze ten, že první vkládá prvky na začátek boxu a druhá na jeho konec.

```
void gtk_box_pack_start(GtkBox *box, GtkWidget *child,
                        gboolean expand, gboolean fill,
                        guint padding);
```

První parametrem je box, do kterého se bude vkládat. Druhý parametr představuje samotný ovládací prvek, který se má do boxu přidat. Třetí parametr pak nabývá hodnoty TRUE nebo FALSE podle toho, jestli mají ovládací prvky společně vyplnit celý přidělený prostor. Čtvrtý parametr určuje, zda má prvek vyplnit celý jemu přidělený prostor. Nastavení tohoto parametru na TRUE má smysl pouze v případě, že předchozí parametr "expand" je nastaven také na hodnotu TRUE.[29] Posledním parametrem se nastavuje v pixelech výplň kolem prvku v boxu.[1] Následující kód zobrazuje možnost, jak lze vytvořit tři ovládací prvky (tlačítko, popisek a text box) a rozmístit je v okně aplikace pomocí boxů.

```
#include <gtk/gtk.h>

int main (int argc, char **argv)

{
 GtkWidget *window;
 GtkWidget *button;
 GtkWidget *text_label;
 GtkWidget *text_entry;
 GtkWidget *box1, *box2, *box3;

 window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
 gtk_window_set_default_size (GTK_WINDOW
                                    (window),500,250);

 button = gtk_button_new_with_label("Klik");
 text_entry = gtk_entry_new();
 text_label = gtk_label_new("Text Text Text");
 box1 = gtk_box_new (GTK_ORIENTATION_HORIZONTAL,20);
 box2 = gtk_box_new (GTK_ORIENTATION_HORIZONTAL,20);
 box3 = gtk_box_new (GTK_ORIENTATION_VERTICAL,20);

 gtk_box_pack_start(GTK_BOX(box1),
                    text_label, TRUE, TRUE, 20);
 gtk_box_pack_start(GTK_BOX(box1),
                    text_entry, TRUE, TRUE, 20);
 gtk_box_pack_start(GTK_BOX(box2),
                    button, TRUE, TRUE, 20);
 gtk_box_pack_start(GTK_BOX(box3), box1,
                    FALSE, FALSE, 20);
 gtk_box_pack_start(GTK_BOX(box3), box2,
                    FALSE, FALSE, 20);

 gtk_container_add(GTK_CONTAINER(window), box3);
 gtk_widget_show_all(window);
 gtk_main ();

 return 0;
}
```

Po přeložení a spuštění programu se zobrazí okno s rozmístěnými ovládacími prvky viz obrázek 7. Na obrázku jsou zakresleny a barevně odlišeny hranice boxů, do kterých byly prvky vloženy.




**Obrázek 7: Rozmístění prvků pomocí boxů**

### 2.8.5 Signály a funkce zpětného volání

Po tom, co jsou v hlavním okně aplikace rozmístěny požadované ovládací prvky, které jsou nezbytné pro práci s aplikací je nutné některým prvkům nadefinovat nějakou akci. Definovaná akce se pak provede jakmile dojde k interakci s daným prvkem. Grafické knihovny musí mít implementovaný určitý mechanismus, který zařídí provedení požadované akce, například pokud dojde ke kliknutí na konkrétní tlačítko apod. Aplikace s grafickým uživatelským rozhraním musí nepřetržitě sledovat a reagovat na vstup od uživatele a na základě toho tak i pracovat. [1]

GTK+ má svůj systém na zpracování událostí, kterému se říká: signály a zpětná volání. Hlavním principem je, že objekt vyvolá nějaký signál a k tomuto signálu je napojena funkce, která je zavolána pokaždé, když dojde k vyvolání daného signálu. Funkcím, které jsou napojeny na zmiňované signály se říká funkce zpětného volání.[1]

Ve výsledku tedy stačí pouze napsat funkce a ty napojit na signál u daného prvku. Předpis funkce zpětného volání je následující:

```
void nazev_fce (GtkWidget *widget, gpointer user_data);
```

Samotná funkce přebírá dva parametry. Jedním z parametrů je ukazatel na objekt, ze kterého byl signál vyslán a druhý je ukazatel, kterým lze do funkce předat nějaká data.

Po napsání funkce zpětného volání už je jen potřeba napojit tuto funkci na konkrétní signál pro daný prvek. To se dělá skrze funkci s názvem *g_signal_connect*, která má následující definici:

```
gulong g_signal_connect(gpointer *object,
                        const gchar *name,
                        GCallback func,
                        gpointer user_data);
```

Prvním parametrem je ovládací prvek, který bude vysílat daný signál, druhým parametrem je název signálu na který se bude reagovat, třetí je funkce zpětného volání a čtvrtý parametr je pak ukazatel na nějaká data, které se mohou předat do funkce zpětného volání. Příklad jednoduché funkce zpětného volání, která může být napojena na klasické tlačítko a která bude vyvolána po každém kliknutí na tlačítko může být zapsána následujícím způsobem:

```
void c_btn_click(GtkWidget *button, gpointer data)
{
      gtk_button_set_label(GTK_BUTTON(button),
                                "Nový popisek");
}
```

Uvnitř funkce je jednoduchý kód, který po kliknutí na tlačítko změní popisek tlačítka na *"Nový popisek"*. Poté už stačí pouze přiřadit k existujícímu tlačítku (zde s názvem button) funkci zpětného volání, která se bude volat pokaždé, jakmile se na tlačítko klikne.

```
g_signal_connect(button,
                 "clicked",
                  G_CALLBACK(c_btn_click),
                  NULL);
```

# 3 Vlastní práce

## 3.1 Aplikace na monitorování stavu systému

Hlavním cílem této části bude vytvoření aplikace, která bude určená pro desktopové prostředí operačního systému Linux a která bude mít za úkol poskytovat základní informace o operačním systému a instalovaném hardwaru. Dále bude také sbírat a zpracovávat informace o využití vybraných systémových zdrojů.

Aplikace bude disponovat grafickým uživatelským rozhraním, které bude vytvořeno za pomoci sady nástrojů GTK+ 3. Využití grafického rozhraní umožní zobrazovat informace v přehlednější a srozumitelnější podobě a také v podobě na který jsou běžní uživatele počítače zvyklí. Informace, které bude aplikace zpracovávat budou zobrazovány skrze jednotlivé její části. Dále zde také bude možnost tyto informace vytisknout buď na tiskárně nebo uložit do souboru například .pdf a vytvořit tak stručný report o stavu sledovaného systému.

Základní požadavky na aplikaci jsou:

- Grafické uživatelské rozhraní
- Přizpůsobování pozice/velikosti ovládacích prvků v okně apliakce v reakci na zmenšení či zvětšení hlavního okna aplikace
- Průběžné sledování a zpracovávání vybraných informací o daném prostředí v pravidelných intervalech
- Zobrazení získaných informací ve srozumitelném a v pro člověka čitelném formátu
- Členění a zobrazení informací v sekcích podle jejich povahy
- Možnost sestavení reportu ze získaných informací a jeho vytisknutí nebo uložení do souboru

### 3.1.1 Informace, které aplikace bude sledovat

Hlavním úkolem aplikace bude shromažďování, zpracovávaní a zobrazování informací, které jsou z pohledu běžného uživatele počítače nejdůležitější a nejčastěji vyžadované. Sbírané informace budou členěny do čtyř okruhů.

První okruh bude obsahovat informace o základním obrazu systému. Budou zde stručné a obecné informace o operačním systému (typ os, název distribuce, verze dané distribuce, verze jádra) a hardwaru počítače (procesor, velikost operační paměti, model grafické karty a model základní desky).

V druhém okruhu bude podrobněji sledována paměť počítače. Konkrétně celková instalovaná velikost operační paměti a její historické využití. Dále zde budou uvedeny detekovaná disková zařízení, jejich kapacita a zaplnění apod.

Ve třetím okruhu bude zase podrobněji sledován procesor. Zde budou hlavní informace o procesoru jako je model procesoru, frekvence, počet jader, ale také sledováno vytížení procesoru za posledních šedesát vteřin. Dále zde také bude sledováno vytížení jednotlivých jader procesoru.

Čtvrtý okru pak bude obsahovat informace o běžících procesech, kde u každého procesu bude zaznamenán jeho název, ID, v jakém stavu se proces nachází a velikost přidělené paměti pro daný procesu.

## 3.2 Návrh vzhledu aplikace

Grafické uživatelské rozhraní je navrženo tak, aby sledované informace, které budou aplikací zpracovávané byly přehledně a srozumitelně zobrazovány. Pro přehlednost jsou získané systémové informace členěny do několika skupin, kde každá skupina obsahuje logicky související druh informací. V grafickém rozhraní jsou tyto skupiny informací zobrazovány skrze celkem čtyři „karty“, kde každá karta poskytne získané informace v přehledném formátu. Jednotlivé karty budou rozděleny následujícím způsobem:

- Karta s obecnými informacemi o systému
- Karta s informacemi o paměti počítače
- Karta s informacemi o procesoru
- Karta s informacemi o spuštěných procesech

V horní části okna aplikace bude umístěné horizontální menu aplikace, které bude obsahovat dvě položky. První položka s názvem *File* (Soubor) bude obsahovat dvě podpoložky. První bude vyvolávat dialog pro tisk informací a druhá ukončovat

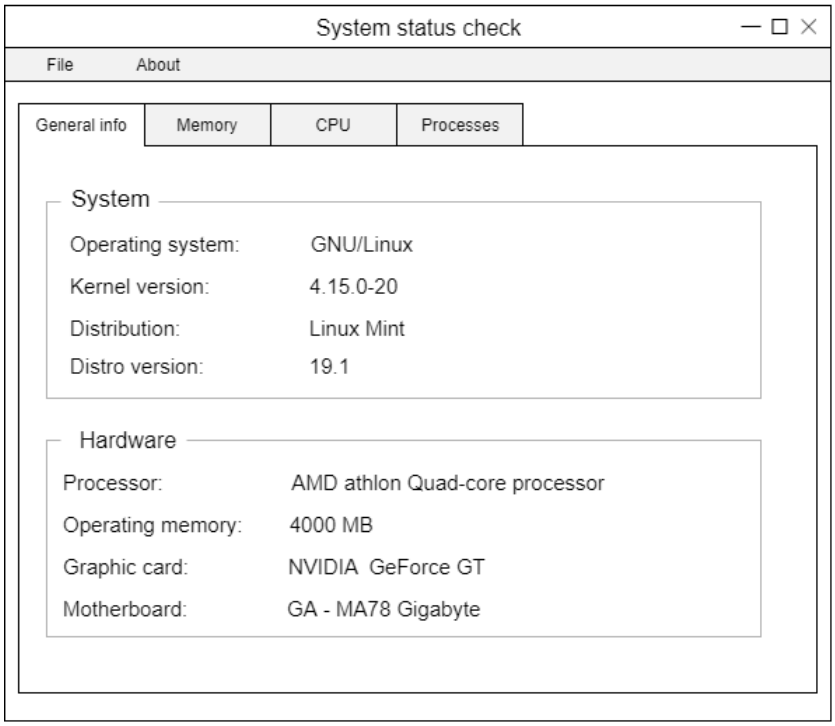
aplikaci. V pořadí druhá položka hlavního menu s popisem About (O aplikaci) bude vyvolávat pouze informativní dialog se základními informacemi o aplikaci.

Před samotným programováním aplikace jsou nejprve pro jednotlivé části aplikace vytvořeny návrhy jejich vzhledu (wireframy), na kterých bude znázorněno rozmístění ovládacích prvků a určeno jaké informace budou na které kartě zobrazovány.

### 3.2.1 Karta s obecnými informacemi o systému

Karta *„General info“* bude obsahovat pouze základní obecné informace o daném prostředí. Tyto informace budou načteny při spuštění aplikace a nebudou po dobu běhu aplikace nijak pozměňovány či aktualizovány, protože to není ani z povahy uvedených informací potřebné.

Pro přehlednější podání informací budou v této kartě zobrazované informace ještě rozděleny do dvou podsekcí. Konkrétně na systémové informace vztahující se k operačnímu systému a na hardwarové informace vztahující se čistě k instalovanému hardwaru daného počítače. Zmíněné podsekce budou od sebe vizuálně odlišeny pomocí rámečků a titulku umístěného v levém horním rohu každého rámečku. Grafický návrh okna aplikace s aktivní kartou pro zobrazení obecných informací je uveden na obrázku 8.

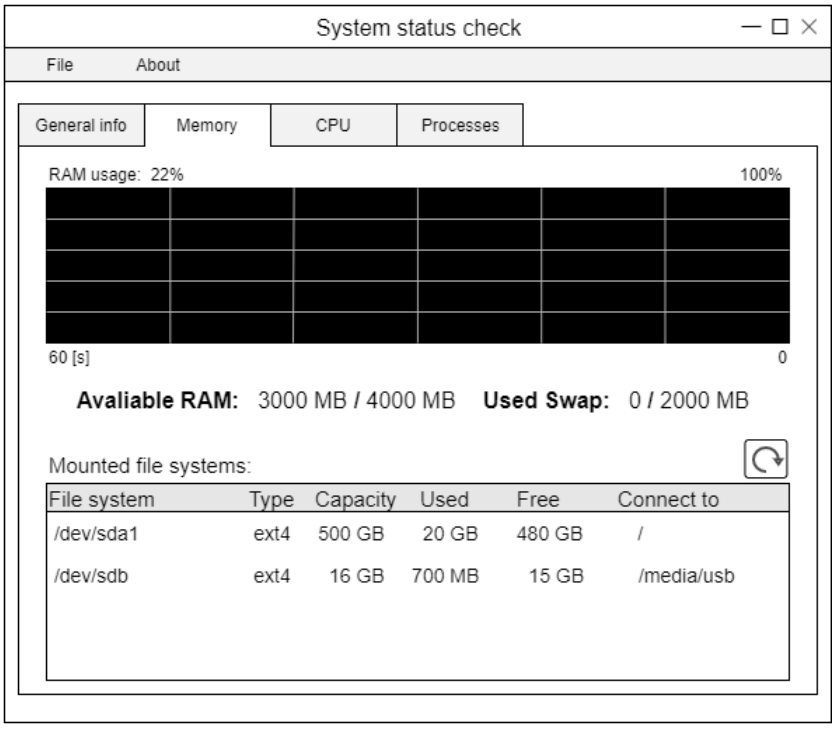



**Obrázek 8: Karta s obecnými informacemi**

### 3.2.2 Karta s informacemi o paměti počítače

V kartě „Memory“ bude uveden celkový přehled o stavu paměti počítače. Konkrétně se tento přehled vztahuje na operační paměť, připojené diskové zařízení a diskové oddíly. Hlavní částí karty je graf pro sledování vývoje využití operační paměti během poslední minuty. Ten je umístěn v horní části karty. Historický vývoj využití operační paměti je zobrazen křivkou v grafu, která je překreslována každou vteřinu a postupně přechází zprava do leva. Úroveň křivky v grafu znázorňuje velikost využití operační paměti k danému časovému okamžiku v procentech. V Grafu na ose X je tedy uveden čas a na ose Y pak velikost využití operační paměti v procentech. Číselné vyjádření aktuálního využití operační paměti v procentech je uvedeno nad levým horním rohem grafu s popiskem „Ram usage“. Pod grafem je ještě zobrazena aktuální dostupná a celková velikost operační paměti v megabytech. Vedle je dále uvedena velikost odkládacího prostoru a kolik z tohoto prostoru je využito.

Ve spodní polovině karty je zobrazen seznam připojených disků a diskových oddílů. Každý řádek v seznamu představuje detekované datové zařízení. Ve sloupcích jsou uvedeny vlastnosti pro daný disk/oddíl. Pro každý disk je sledován jeho název, typ souborového systému, celková kapacita, využité místo, volné místo a kam je dané zařízení připojeno. Vpravo nad seznamem je umístěno tlačítko, které pak zaktualizuje seznam.

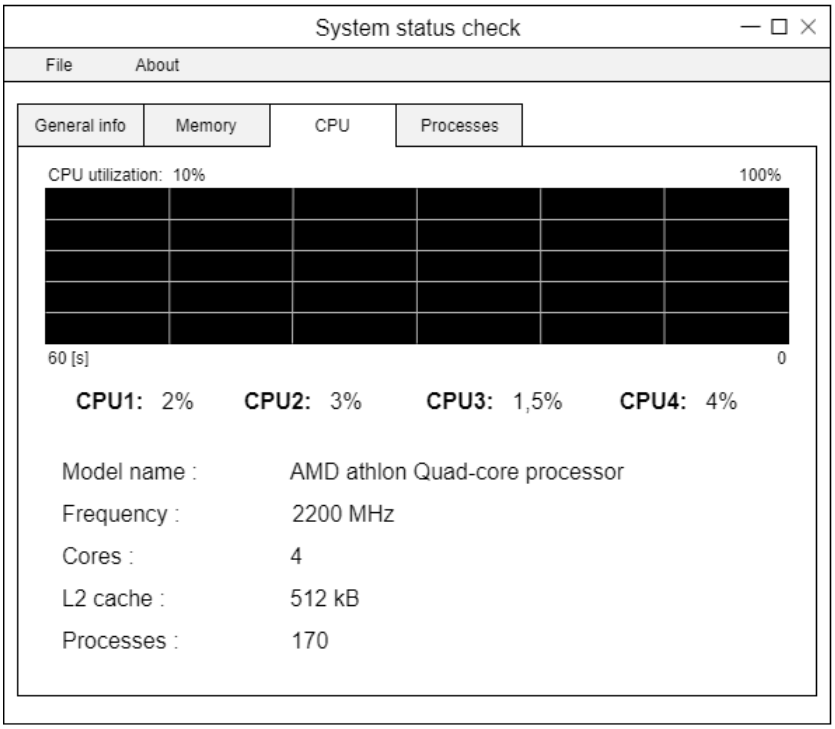



**Obrázek 9: Karta s informacemi o paměti**

### 3.2.3 Karta s informacemi o procesoru

V kartě "CPU" je zobrazen detailnější pohled na procesor počítače. Zde jsou uvedeny hlavní parametry procesoru a jeho vytížení systémem. Grafický návrh karty aplikace s informacemi o procesoru je zobrazen na obrázku 10. Obdobně jako u karty s informacemi o paměti počítače je hlavní částí karty graf umístěný v horní části okna, který informuje o vytížení procesoru během poslední uplynulé minuty. V grafu je na ose X uveden čas, který zobrazuje interval 0 až 60 vteřin. Na ose Y se pak zobrazuje vytížení procesoru v procentech. Zobrazení vytížení procesoru v grafu je realizováno prostřednictvím křivky, která se překresluje každou vteřinu a postupuje zprava doleva, kde více vpravo jsou novější hodnoty z hlediska času. Aktuální hodnota vytížení procesoru v procentech je také uvedena v číselné podobě v levé horní části nad grafem s popiskem "CPU utilization".

Pod grafem pak jsou vypsány základní parametry procesoru, jako je model procesoru, jeho frekvence, počet jader, velikost L2 cache a celkový počet spuštěných procesů. Protože dnešní procesory disponují více jádry, je zde také ještě pro každé jádro zobrazena hodnota reprezentující jeho vytížení v procentech. Každé jádro je zde označeno jako CPUX, kde X je celé číslo charakterizující konkrétní jádro.

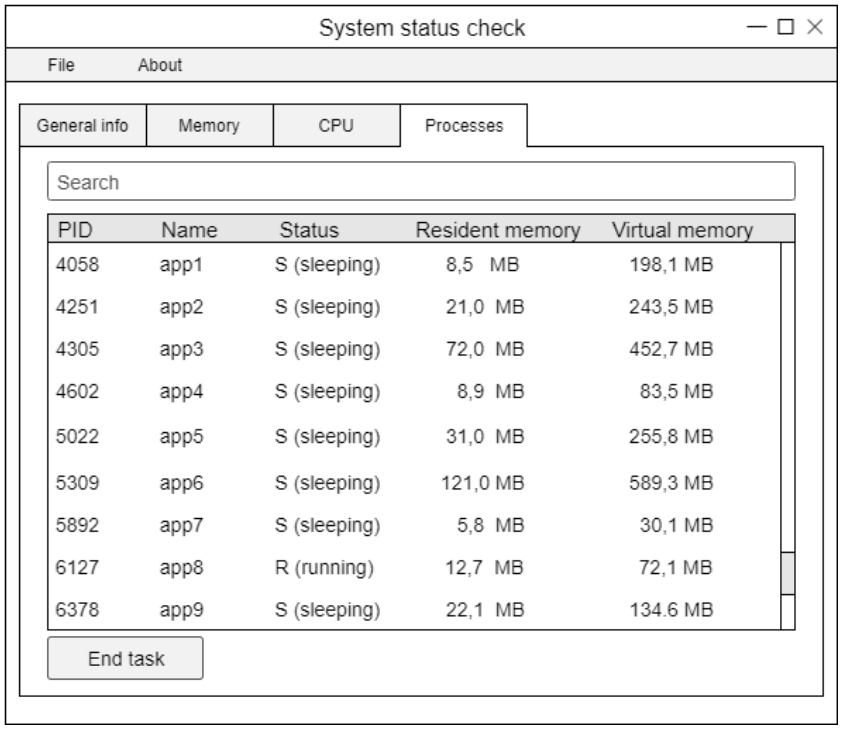



**Obrázek 10: Karta s informacemi o procesoru**

### 3.2.4 Karta s informacemi o spuštěných procesech

V kartě "Processes" je zobrazen seznam všech procesů, které jsou aktuálně spuštěny pod operačním systémem. Ke každému procesu jsou pak ještě uvedeny doplňující informace. Princip zobrazení seznamu procesů je realizován prostřednictvím tabulky, kde každý řádek představuje jeden spuštěný proces. Ve sloupci tabulky jsou pak uvedeny sledované vlastnosti pro každý proces. U každého procesu je sledováno jeho Process ID (PID), název procesu, status ve kterém se proces právě nachází a velikost rezidentní a virtuální paměti. Protože na linuxových systémech bývá spuštěno takové množství procesů, které se pohybuje v řádu stovek, je obsah tabulky/seznamu vložen do rolovacího okna pro přehledné zobrazení všech načtených procesů.

Dále také z důvodu většího množství spuštěných procesů je nad tabulkou s procesy umístěno vyhledávací pole rozpínající se přes celou šířku karty pro možnost vyhledání konkrétního procesu. Vyhledání procesu proběhne na základě alespoň částečné shody řetězce zapsaného do vyhledávacího pole s názvem procesu. Každý proces v tabulce lze vybrat/označit a takto vybraný proces je možné ukončit pomocí tlačítka s názvem "End task", které se umístěné vlevo pod tabulkou se seznamem procesů. Grafický návrh karty s informacemi o procesech je zobrazen na obrázku 11.




**Obrázek 11: Karta s informacemi o procesech**

## 3.3 Struktura kódu

Po návrhu vzhledu aplikace je následně přistoupeno k psaní zdrojového kódu. Pro psaní a uložení kódu do souboru je použit klasický textový editor, který bývá běžně dodáván s téměř všemi linuxovými distribucemi. Samotný zdrojový kód je členěn a psán do čtyř zdrojových souborů v závislosti na jeho povaze, plus jeden hlavičkový soubor. Tímto přístupem bude probíhat lepší správa napsaného kódu a bude i tak oddělen kód, který má za úkol zajišťovat získávání dat ze systému pro následné zpracování zpracování od kódu pro vytvoření grafického rozhraní. Tyto soubory budou nakonec pomocí programu *make* slinkovány dohromady a vytvořen tak výsledný spustitelný program.

Názvy a význam jednotlivých zdrojových souborů:

- **app_gtk.h** – nejprve je definován hlavičkový soubor, kde jsou připojeny všechny potřebné knihovny a který je pak vložen do ostatních souborů. Dále jsou zde definovány vlastní uživatelské datové typy a nakonec zapsány hlavičky vytvářených funkcí v programu, aby byly viditelné i pro ostatní zdrojové soubory.

- **main.c –** tento soubor obsahuje pouze hlavní funkci main, ve které je zavolána funkce pro vytvoření a zobrazení hlavního okna aplikace a funkce pro inicializaci hodnot základních proměnných.

- **interface.c –** do souboru interface.c je psán kód, který se stará o vytvoření grafického rozhraní aplikace. Jsou zde funkce pro vytvoření hlavního okna aplikace, jednotlivých karet a dalších grafických prvků.

- **callback.c –** soubor callback.c obsahuje funkce pro zpětné volání. Tyto funkce se provedou na základě vygenerovaného signálu daného ovládacího prvku, na který jsou tyto funkce napojeny. Spolu s funkcemi pro zpětné volání jsou zde uloženy ještě časové funkce pro aktualizaci sbíraných dat, které jsou volány vždy v určitém časovém intervalu.

- **data.c –** v tomto souboru jsou napsané funkce pro získávání dat, případně pro nějakou další práci s daty. Tyto funkce získávají potřebná data ze systému, následně je zpracují a uloží do určených proměnných.

## 3.4 Zdroj dat a způsob jejich zpracování

Hlavním zdrojem dat, ze kterého bude aplikace převážně čerpat je virtuální souborový systém s názvem */proc*, který poskytuje mnoho různých informací o systému. Dají se zde nalézt informace o operační paměti, procesoru, ale především veškeré informace o všech spuštěných procesech. Velká část souborů je v tomto souborovém systému běžně dostupná pro čtení a není tedy pro získání požadovaných dat potřeba žádné speciální oprávnění. Tento virtuální souborový systém (konkrétně jeho obsah) v podstatě odráží stav celého systému. Funkce pro získávání dat z vybraných zdrojů jsou napsané v souboru data.c. Celkem je pro načítání dat ze systému napsáno několik funkcí. Pro ukládání získaných informací jsou v hlavičkovém souboru *app_gtk.h* definované speciální uživatelské datové typy pomocí struktur.

Pro uchování informací o procesoru po dobu běhu programu je vytvořen uživatelský datový Cpu. Tento typ je schopen udržovat následující druh informací:

```
typedef struct {
   char nazev[50];                  //model procesoru
   char cache_size[15];             //velikost L2 cache
   float frekvence;                 //frekvence procesoru
   int num_of_cores;                //počet jader
   float history_utilization[61];//historie vytížení CPU
   int history_index;               //proměnná pro indexaci
   Core core[24];                   //jádra CPU
}Cpu;
```

Aby bylo možné přehledně uchovávat i informace o jednotlivých jádrech procesoru, je pro tento účel vytvořen uživatelský datový typ Core:

```
typedef struct {
   float utilization;               //vytížení jádra v %
   long int total_sum;              //suma časů stavů jádra
   long int idle;                   //čas idle
}Core;
```

Pro uložení informací o spuštěných procesech slouží datový typ Proc:

```
typedef struct {
  char name[100];                   //název procesu
  int pid;                          //id procesu
  char status[20];                  //status procesu
  char memory[10];                  //rezidentní paměť
  char memoryVm[10];                //virtuální paměť
}Proc;
```

Pro ukládání informací o stavu operační paměti je vytvořen datový typ Ram, který má následující definici:

```
typedef struct {
  float totalRam;                 //celková kapacita RAM
  float avRam;                    //dostupná kapacita RAM
  float swapTotal;                //celková kapacita swap
  float swapFree;                 //dostupná kapacita swap
  float history_usage[61];        //historie využití RAM
  int history_index;              //proměnná pro indexaci
}Ram;
```

Pro uložení informací o připojených discích a diskových oddílech slouží vytvořený datový typ Disk:

```
typedef struct {
  char disk_name[50];             //název filesystému
  char type[20];                  //typ filesystému
  char capacity[20];              //celková kapacita
  char used_space[20];            //využité místo
  char free_space[20];            //volné místo
  char connect_to[100];           //místo připojení
}Disk;
```

Hlavní struktura, která bude sjednocovat získané informace a vytvářet tak jeden pohled na celý systém se jmenuje *swhw_detail*. S definicí struktury je rovnou vytvořena jedna proměnná typu *struct swhw_detail* a ukazatel `*swhw_ptr` skrze který se pak bude k položkám struktury přistupovat. Struktura má následující definici:

```
struct swhw_detail{
  char os[50];                    //název operačního systému
  char kernelRelease[50];         //verze jádra
  char distro[50];                //název distribuce
  char v_distro[50];              //verze distribuce
  char g_adapter[100];            //model grafické karty
  char board[100];                //model základní desky
  int cnt_proc;                   //počet spuštěných procesů
  int cnt_disk;                   //počet připojených disků
  Proc proc[500];                 //Pole pro procesy
  Disk disk[20];                  //pole pro disky
  Cpu cpu;                        //procesor
  Ram ram;                        //operační paměť
}swhw_d, *swhw_ptr;
```

### 3.4.1 Výpočet hodnot vytížení procesoru

První z pěti funkcí ze souboru data.c, která se stará o získávání dat je funkce s názvem *cpu_utilization()*. Úkolem této funkce je získat, vypočítat a uložit celkovou hodnotu vytížení procesoru v procentech. Dále je také výpočet vytížení proveden pro každé jádro procesoru zvlášť.

Hodnoty pro výpočet vytížení procesoru jsou načítány a vypočítány z údajů nacházející se v souboru */proc/stat*, který obsahuje určité systémové statistiky. Pro výpočet vytížení procesoru je v tomto souboru důležitých prvních N řádků (počet načítaných řádků je závislý na počtu jader procesoru). Každý z načítaných řádků zobrazuje množství času, které procesor strávil v různých stavech. Úplně první řádek agreguje hodnoty za všechny jádra a tudíž představuje procesor jako celek. Ostatní načítané řádky už reprezentují konkrétní jádra procesoru. Požadované řádky začínají řetězcem "cpu", pokud se jedná o statistiku konkrétního jádra, je za tento řetězec napsáno ještě celé číslo odlišující jednotlivá jádra: cpu0, cpu1... Za tímto řetězcem následuje deset číselných údajů. Struktura každého řádku má následující tvar:

```
cpu[N] user nice system idle iowait irq softirq steal quest quest_nice
```

Jednotlivé hodnoty mají podle manuálových stránek následující význam:

| | |
|---|---|
| `user` | - čas strávený v uživatelském režimu |
| `nice` | - čas strávený v uživatelském režimu s nízkou prioritou |
| `system` | - čas strávený v systémovém režimu |
| `idle` | - čas strávený v nečinném stavu |
| `iowait` | - čas čekání na dokončení I/O operací |
| `irq` | - čas strávený obsluhou přerušení |
| `softirq` | - čas strávený obsluhou softirgs |
| `steal` | - čas strávený v ostatních operačních systémech, které běží ve virtualizovaném prostředí |
| `quest[_nice]` | - čas strávený během virtuálního CPU pro hostující operační systémy |

Hodnotu vytížení procesoru/jádra z výše uvedených hodnot je nutné následně dopočítat. Pro výpočet vytížení procesoru/jádra je nutné načíst hodnoty ve dvou různých časových okamžicích (v této aplikaci je rozmezí časových okamžiků nastaveno na jednu vteřinu). Výpočet spočívá v tom, že se udělá suma všech hodnot (časů) daného řádku, zvlášť je nutné si ještě zapamatovat hodnotu idle. Při dalším čtení se provedou stejné operace. Výpočet pak proběhne podle následujícího vzorce:

$$vytížení\ (\%) = \frac{suma - předchozí\_suma - idle - předchozí\_idle}{suma - předchozí\_suma} * 100$$

Uvnitř funkce *cpu_utilization()* je nejdříve otevřen soubor /proc/stat pro čtení. Pokud se soubor podařilo úspěšně otevřít, postupně se načítají pomocí funkce *fgets* jednotlivé řádky jako text. Každý řádek je po jeho načtení funkcí *strtok* rozdělen na tzv. tokeny, což jsou už požadované hodnoty potřebné pro výpočet. Je nutné si zapamatovat dvě hodnoty. Zvlášť se uloží hodnota idle a součet všech hodnot daného řádku. Dále se podle výše uvedeného vzorce vypočítá celkové vytížení procesoru i vytížení jednotlivých jader. Získané hodnoty se pak uloží pro příští výpočet. Popsaný princip vykonává následující část kódu funkce:

```
stat = fopen("/proc/stat", "r");
if(stat != NULL)
{
  for(i=0; i <= swhw_ptr->cpu.num_of_cores; i++)
  {
    fgets(str, 200, stat);
    k = 0;
    for(token=strtok(str," ");token!=NULL;token=strtok(NULL, " "))
    {
      if(k > 0){
         sum_times += atoi(token);
         if(k == 4)
            idle = atoi(token);
        }
        k++;
    }
    if(swhw_ptr->cpu.history_index > 0)
    {
     sumDiff = sum_times - swhw_ptr->cpu.core[i].total_sum;
     idleDiff = idle - swhw_ptr->cpu.core[i].idle;
     swhw_ptr->cpu.core[i].utilization =
      ((sumDiff - idleDiff)/(sumDiff)) * 100.0;
    }
    swhw_ptr->cpu.core[i].idle = idle;
    swhw_ptr->cpu.core[i].total_sum = sum_times;
    sum_times = 0;
  }fclose(stat);
}
```

### 3.4.2 Získání informací o RAM

Všechny potřebné informace tykající se operační paměti jsou získány ze souboru *meminfo*, který je také v adresáři */proc*. Každý údaj o operační paměti je v tomto souboru na samostatném řádku. Každý řádek ale začíná nejprve názvem vyjadřující, co daná hodnota znamená. Název je následovaný dvojtečkou a několika mezerami. Pak je teprve

uvedená hodnota v kilobytech (například pro celkovou velikost operační paměti: *"MemTotal: 8000000 kb")*. V tomto souboru jsou z pohledu aplikace důležité čtyři řádky/údaje. Zejména jsou z tohoto souboru získány údaje o celkové a dostupné velikosti operační paměti a celkové a nevyužité velikosti swap prostoru.

Funkce, která načítá údaje o RAM se jmenuje *ram_usage()*. Uvnitř těla funkce je nejprve otevřen soubor *meminfo* v režimu pro čtení. Poté je soubor procházen řádek po řádku pomocí funkce *fgets* a hledány řádky s požadovaným názvem. Pro nalezení řádku s požadovanou hodnotou je vytvořená funkce *checkSubstring*, která zkontroluje zda načtený řádek začíná odpovídajícím názvem. Číselný údaj o paměti je z daného řádku vybrán pomocí funkce *strtok*, který je poté ještě převeden na megabyty a uložen do příslušných proměnných. Úplně na konec je vypočítána hodnota celkového využití operační paměti v procentech, kterou pak funkce vrací. Část kódu funkce je následující:

```
stat = fopen("/proc/meminfo", "r");
if(stat != NULL)
{
   while(fgets(str, 100, stat) != NULL)
   {
       token = strtok(str, " ");
       token = strtok(NULL, " ");

       if(checkSubString(str, "MemTotal"))
            swhw_ptr->ram.totalRam = (atof(token)/1000.0);
       else if(checkSubString(str, "MemAvailable")
            swhw_ptr->ram.avRam = (atof(token)/1000.0);
       else if(checkSubString(str, "SwapTotal"))
            swhw_ptr->ram.swapTotal = (atof(token)/1000.0);
       else if(checkSubString(str, "SwapFree"))
            swhw_ptr->ram.swapFree = (atof(token)/1000.0);
  }
  fclose(stat);
  ramUsage=(1-swhw_ptr->ram.avRam/swhw_ptr->ram.totalRam)*100.0;
}
```

### 3.4.3 Získání informací o procesech

Data ke všem spuštěným procesům lze najít také v adresáři */proc*. Každý proces zde reprezentuje jedna složka, která je pojmenovaná číslem představující ID procesu. V každé složce jednoho procesu je několik různých souborů. Pro získání potřebných dat stačí získat údaje ze souboru proc/[PID]*/status*, kde lze najít id, název, status procesu a některé informace ohledně přidělené paměti.

Funkce, která zajišťuje sběr informací o všech spuštěných procesech je pojmenována *get_process()*. Princip funkce spočívá v tom, že nejprve otevře adresář /proc a následně iteruje přes všechny obsažené objekty. Postupně se hledají všechny adresáře s číselným pojmenováním reprezentující spuštěné procesy. Pokud je nalezen adresář s číselným pojmenováním, je v tomto adresáři otevřen pro čtení soubor /proc/[PID]/status a v něm jsou hledány řádky s požadovanými údaji. Získané údaje jsou následně zpracovány a uloženy do odpovídajících proměnných. O zmíněnou funkcionalitu se stará následující část kódu:

```
struct dirent *de;
DIR *dr = opendir("/proc");
while((de = readdir(dr)) != NULL)
{
  if(de->d_type == DT_DIR && isdigit(de->d_name[0]))
  {
    path = g_strdup_printf("/proc/%s/status", de->d_name);
    stat = fopen(path, "r");
    if(stat != NULL){
     while(fgets(str, 100, stat) != NULL){
       del_new_line(str, strlen(str), 0);
       token = strtok(str, "\t");
       token = strtok(NULL, "\t");

       if(checkSubString(str, "Name:"))
          strncpy(swhw_ptr->proc[k].name, token, 100);
       else if(checkSubString(str, "Pid:"))
          swhw_ptr->proc[k].pid = atoi(token);
       else if(checkSubString(str, "State:"))
          strncpy(swhw_ptr->proc[k].status, token, 20);
       else if(checkSubString(str, "VmRSS:")){
          vmsize = token;
          token = strtok(vmsize, " ");
          snprintf(swhw_ptr->proc[k].memory,
              10 ,"%.1f MB", atof(token)/1000.0);
      }
      else if(checkSubString(str, "VmSize:")){
          vmsize = token;
          token = strtok(vmsize, " ");
          snprintf(swhw_ptr->proc[k].memoryVm,
            10 ,"%.1f MB", atof(token)/1000.0);
      }
    } fclose(stat);
   }
      k++;
      g_free(path);
  }
 } closedir(dr);
```

### 3.4.4 Získání informací o připojených discích

Informace týkající se připojených disků a diskových oddílů zpracovává funkce s názvem *get_disk()*. Potřebné informace lze získat z výstupu shellového příkazu df -HT, který poskytne informace o názvu, typu filesystému, kapacity, zaplnění a místa připojení. Tento příkaz lze z programu spustit v subprocesu pomocí funkce *popen* a výstup pak zpracovávat stejným způsobem jako se zpracovává výstup při čtení klasického souboru.

Každý řádek ve výstupu reprezentuje jeden připojený filesystem, kde ve sloupcích jsou uvedeny vlastnosti pro daný řádek (disk/oddíl). Výstup je tedy zpracováván postupně po řádcích funkcí *fgets* a každý řádek je na potřebné části rozdělen funkcí *strtok*. Kód funkce je následující:

```
stat = popen("df -HT --exclude-type=tmpfs --exclude-type=devtmpfs
                | tail -n +2", "r");
if(stat != NULL)
{
 while(fgets(str, 100, stat) != NULL){
    del_new_line(str, strlen(str), 0);
    i = 0;
 for(token = strtok(str, " ");token!=NULL;token=strtok(NULL, " "))
 {
    if(i == 0)
       strncpy(swhw_ptr->disk[k].disk_name, token, 50);
    else if(i == 1)
       strncpy(swhw_ptr->disk[k].type, token, 20);
    else if(i == 2)
       strncpy(swhw_ptr->disk[k].capacity, token, 20);
    else if(i == 3)
       strncpy(swhw_ptr->disk[k].used_space, token, 20);
    else if(i == 4)
       strncpy(swhw_ptr->disk[k].free_space, token, 20);
    else if(i == 6)
       strncpy(swhw_ptr->disk[k].connect_to, token, 100);
    i++;
  } k++;
 } pclose(stat);
} swhw_ptr->cnt_disk = k;
```

### 3.4.5 Získání obecných informacích o systému

Funkce popsané v předchozích částech jsou během programu využívány opakovaně, protože je nezbytné informace získané těmito funkcemi po dobu běhu

programu aktualizovat. Aktualizace informací probíhá buď automaticky po uplynutí stanoveného času nebo po vykonání nějaké uživatelské akce.

Ostatní funkce ze souboru data.c, které také sbírají informace jsou využity jen při spuštění aplikace k načtení základních parametrů, které už za běhu programu nebude potřeba aktualizovat. Jedná se zejména o informace o typu a verzi operačního systému či o parametry instalovaného hardwaru.

O načtení informací o procesoru se stará funkce s názvem *get_cpuInfo*, která bere informace ze souboru /proc/cpuinfo, odkud jsou získány údaje o modelu procesoru, počtu jeho jader, velikosti L2 cache a frekvenci. Tato funkce zpracovává data obdobným způsobem jako předchozí uvedené funkce. Nejprve se tedy otevře soubor *cpuinfo* pro čtení a následně funkcí se *fgets* načítá řádek po řádku, ze kterého se pak vyberou potřebné údaje a uloží do připravených proměnných. V souboru je řada dalších informací, ale pro program jsou všechny potřebné údaje umístěné do cca čtrnáctého řádku, takže stačí číst soubor pouze do této pozice a pak čtení ukončit. Velmi důležitá hodnota získaná z tohoto souboru je počet jader procesoru, od které se pak odvíjí další operace například načítání počtu řádků ze souboru */proc/stat* odkud se berou hodnoty pro vypočet vytížení jednotlivých jader procesoru apod. Funkce přijímá jako parametr ukazatel na soubor, ze kterého bude číst. Kód funkce je následující:

```
void get_cpuInfo(FILE *stat)
{
      char str[100];
      char *token;
      int numLines = 0;

      while(fgets(str, 100, stat) != NULL && numLines < 14 ) {
            del_new_line(str, strlen(str), 0);
            token = strtok(str, ":");
            token = strtok(NULL, ":");

            if(checkSubString(str, "model name"))
                  strncpy(swhw_ptr->cpu.nazev, token, 50);
            else if(checkSubString(str, "cpu MHz"))
                  swhw_ptr->cpu.frekvence = atof(token);
            else if(checkSubString(str, "cpu cores"))
                swhw_ptr->cpu.num_of_cores = atoi(token);
            else if(checkSubString(str, "cache size"))
                  strncpy(swhw_ptr->cpu.cache_size, token, 15);
            numLines++;
      }
}
```

Obdobným způsobem jsou postavené i zbývající funkce s názvem *get_distroInfo*, *get_osInfo*, *get_graphicInfo* a *get_boardInfo*. První dvě uvedené funkce se postarají o získání typu operačního systému, verzi jádra, typu a verzi distribuce. Třetí pak načítá typ grafické karty a čtvrtá typ a výrobce základní desky.

Před vytvořením hlavního okna aplikace ve funkci *main* je ještě zavolána funkce s názvem *initHwSwvalue*. Uvnitř této funkce jsou postupně otevřeny potřebné soubory k inicializaci základních hodnot proměnných a volány funkce *get_cpuInfo*, *get_distroInfo*, *get_osInfo*, *get_graphicInfo* a *get_boardInfo*. Funkcím je předáván ukazatel na otevřený soubor, který mají za úkol zpracovat. Pokud by byl problém se zpracováním některého ze souborů, nastaví se odpovídající proměnné, do kterých se mají potenciální informace uložit, na defaultní hodnoty.

## 3.5 Vytvoření grafického uživatelského rozhraní

Na začátku tvorby grafického rozhraní je nejprve nutné vytvořit hlavní okno aplikace, do kterého budou následně přidávány další ovládací prvky. Jak již bylo řečeno, funkce pro vytvoření grafického rozhraní jsou umístěné v souboru *interface.c*. V tomto souboru je celkem napsáno jedenáct funkcí. Na úplný začátek souboru je vložen pomocí direktivy include odkaz na hlavičkový soubor *app_gtk.h* a vytvořen ukazatel, který bude ukazovat na hlavní okno aplikace.

### 3.5.1 Hlavní okno aplikace

Pro vytvoření hlavního okna aplikace je napsaná funkce *create_window()*, ve které jsou nadefinovány základní parametry hlavního okna, menu aplikace a kontejner, do kterého jsou vkládány jednotlivé stránky/karty, kde budou vhodným způsobem zobrazovány získané informace. Tato funkce nepřijímá žádné parametry a nakonec vrací na nově vytvořené okno ukazatel.

```
GtkWidget *create_window();
```

Po zavolání funkce *create_window()* je hlavní okno aplikace vytvořeno zavoláním *gtk_window_new* a jako parametr ji je předána hodnota definující, jakého typu má nově vytvořené okno být. V tomto je předána hodnota GTK_WINDOW_TOPLEVEL, která

říká, že má být vytvořeno klasické okno s rámečkem. Dále je zde vytvořeno základní menu aplikace. Menu je realizováno pomocí widgetu typu GtkMenuBar, který je vytvořen funkcí *gtk_menu_bar_new* a do kterého se pak přidávají jednotlivé položky a podpoložky menu. Každá položka menu je vytvořena funkcí *gtk_menu_item_new_with_label* a následně nastavena tak, aby dohromady vytvářely požadovanou rozbalovací nabídku aplikace.

Nakonec je do okna aplikace přidán widget s názvem notebook. Tento widget je vytvořen pomocí funkce *gtk_notebook_new* a představuje jakýsi kontejner, do kterého jsou vkládány stránky/karty. Pomocí notebooku je zobrazovací část aplikace rozdělena do několika karet, kde každá karta obsahuje jiný obsah. Mezi kartami lze jednoduše přepínat skrze kliknutím na její název umístěný v levém horním rohu notebooku. Každá karta je do notebooku přidána pomocí funkce *gtk_notebook_append_page*, které je předán jako parametr notebook, do kterého má být karta přidána, pak prvek typ GtkWidget, který má být obsahem dané karty a nakonec ještě název/označení přidávané karty. Například přidání karty s informacemi o procesoru do notebooku je v programu provedeno následujícím způsobem

```
gtk_notebook_append_page(GTK_NOTEBOOK(notebook),
                         createCpuPage(),
                         cpuLabel);
```

### 3.5.2 Vytvoření jednotlivých karet aplikace

Pro přehlednost jsou jednotlivé karty, které jsou vkládány do notebooku vytvářeny speciálními funkcemi. Uvnitř funkcí jsou definovány a nastaveny potřebné grafické prvky pro danou karty a ty jsou nakonec vloženy do kontejneru typu GtkBox, na který pak funkce vrací ukazatel.

**Karta General info.**

Funkce, která vytvoří obsah pro první stránku/kartu se jmenuje *createInfoPage*. Zde jsou vytvořené a rozmístěné grafické prvky nesoucí informace o operačním systému a instalovaném hardwaru počítače.

```
static GtkWidget *createInfoPage();
```

Zobrazované informace jsou zde rozděleny ještě do dvou podsekcí v závislosti na tom, jestli se jedná o informace o operačním systémy nebo hardwaru. Zmíněné podsekce jsou

realizovány widgetem typu GtkFrame, který je vytvořen funkcí *gtk_frame_new*. Ve své podstatě se jedná pouze o kontejner, do kterého jsou vloženy prvky typu GtkLabel zobrazující potřebné informace. Při výsledném zobrazení je pak každá podsekce ohraničena rámečkem s titulkem umístěným v levém horním rohu rámečku.

**Karta Memory**

Funkce *createMemoryPage* vytvoří kartu s grafickými prvky pro zobrazení informací o paměti počítače.

```
static GtkWidget *createMemoryPage();
```

Po zavolání funkce je vytvořen widget typu GtkDrawinArea, který představuje plochu, na kterou se bude vykreslovat graf zobrazující historii využití RAM. Takto vytvořená kreslící oblast je poté napojena na funkci zpětného volání *draw_callback_ram*, která se postará o vykreslení grafu.

Dále se vytvoří pole popisků, ve kterých bude zobrazována aktuální hodnota využití RAM, její volná kapacita a hodnota využité velikosti swap prostoru. K aktualizování zobrazených statistik slouží funkce *repaint_ram*, ve které jsou aktualizovány hodnoty pro jednotlivé popisky a také hodnoty pro vykreslení grafu. Hodnoty jsou aktualizovány každou vteřinu pomocí funkce *g_timeout_add*, které je nastaven interval jak často má být funkce *repaint_ram* volána. Dále ji je ještě předán odkaz na pole popisků, jejichž obsah má být aktualizován.

Zobrazení seznamu připojených disků případně jejich oddílů a jejich vlastností je realizováno pomocí widgetu typu GtkTreeView. Tento widget zobrazuje data ve formátu podobném tabulce. Každá řádek v seznamu/tabulce představuje jeden detekovaný filesystém a ve sloupcích jsou pak uvedeny jeho vlastnosti. Pro vytvoření takového seznamu jsou napsány ještě dvě pomocné funkce. První z nich je funkce *new_model_filesystem*, která definuje rozhraní pro vytvořený widget typu GtkTreeView. Druhá funkce s názvem *add_columns_filesystem* pak definuje a přidá do widgetu typu GtkTreeView jednotlivé sloupce. Pro aktualizování seznamu, je vytvořeno ještě tlačítko, které je napojené na funkci zpětného volání refresh_disk, která provede samotnou aktualizaci.

### Karta CPU

Další funkcí pro vytvoření karty aplikace je funkce *createCpuPage*, která vytvoří potřebný grafický vzhled pro zobrazení informací o procesoru a jeho průběžném vytížení.

```
static GtkWidget *createCpuPage();
```

Zavoláním funkce je vytvořen widget typu GtkDrawingArea, který slouží jako podklad pro vykreslení grafu zobrazující historii vytížení procesoru za poslední minutu. O vykreslování grafu se stará funkce zpětného volání *draw_callback_cpu* na kterou je kreslící oblast napojena.

Dále se vytvoří několik popisků pro zobrazení hodnot o vytížení procesoru a jeho jader, počtu spuštěných procesů a pro zobrazení základních parametrů procesoru. Některé zobrazované hodnoty je nutné aktualizovat v pravidelných intervalech. Zejména je nutné pravidelně aktualizovat hodnoty potřebné pro překreslení grafu a hodnoty v popiscích, které zobrazují číselnou informaci o vytížení cpu/jader a počtu spuštěných procesů. O aktualizaci potřebných hodnot se stará funkce *repaint_cpu*, která je volána každou vteřinu pomocí funkce *g_timeout_add*.

### Karta Proccesses

Poslední funkce pro vytvoření karty přidané do notebooku nese název *createProcessPage*. Tato funkce vytvoří vhodné prostředí, které bude schopno zobrazit potřebné informace o spuštěných procesech.

```
static GtkWidget *createProcessPage();
```

Uvnitř funkce jsou vytvořeny tři hlavní ovládací prvky. Prvním z nich je widget typu GtkEntry, pomocí kterého je možné vyhledat v seznamu konkrétní proces podle zadaného názvu. Zobrazení seznamu spuštěných procesů a jejich vlastností je realizováno pomocí widgetu typu GtkTreeView, který zobrazí data ve formátu podobném tabulce. Pro definování rozhraní pro widget GtkTreeView je napsána ještě pomocná funkce *new_model* a pro přidání jednotlivých sloupců do GtkTreeView zase funkce *add_columns*. Nakonec je ještě vytvořeno tlačítko, které ukončí vybraný proces ze zobrazeném seznamu. Tlačítko je napojené na funkci zpětného volání *quit_proces*, která je volána jakmile dojde ke kliknutí na tlačítko a uvnitř které je implementován mechanismus pro ukončení zvoleného procesu.

**Pomocné funkce při vytváření grafického rozhraní**

Zbývající dvě funkce ze souboru interface.c pomáhají s formátováním a lepším zobrazením určitých prvků v okně aplikace. První funkce *create_graph* pouze pomáhá naformátovat a do vytvořit prostředí pro graf umístěný v kartách CPU a Memory. První ze čtyř předaných parametrů funkci jsou textové popisky pro osy a popisek grafu. Ty budou ve funkci vhodně umístěné kolem vykreslovací plochy. Poslední parametr představuje samotnou kreslící plochu, kolem které se mají zmíněné popisky umístit.

```
GtkWidget *createGraph(gchar *xy_start, gchar *y_end,
                       gchar *x_end, GtkWidget *title,
                       GtkWidget *area);
```

Druhá funkce *addLabelRow* vytváří formátovaný popisek. Přebírá dva textové a jeden celočíselný parametr. Po zavolání funkce je vytvořen widget typu GtkLabel, kterému je ze zadaných parametrů nastaven a naformátován obsah. Nakonec funkce na vytvořený widget vrací ukazatel.

```
GtkWidget *addLabelRow(gchar *leftL, gchar *rightL,
                       int spacing);
```

## 3.6 Funkce main

Nakonec je v souboru main.c vytvořena hlavní funkce *main*. V té se nejprve inicializují knihovny GTK+ a zavolá se *initHwSwvaulue()* pro nastavení hodnot základních proměnných. Následně je už zavoláním *create_window()* vytvořeno hlavní okno aplikace, které je následně voláním *gtk_widget_show_all* zobrazeno.

```
#include "app_gtk.h"

gint main(gint argc, gchar *argv[])
{
      gtk_init (&argc, &argv);
      GtkWidget *window;

      initHwSwvalue();

      window = create_window();
      gtk_widget_show_all(window);

      gtk_main();
      return 0;
}
```

## 3.7 Vytvoření reportu ze získaných informací

Pro zrealizování tiskového procesu je použito vysokoúrovňové API GtKPrintOperation. K vyvolání tiskového dialogu pro tisk získaných informací dojde aktivací položky "Print", která je zabalená pod hlavním menu "File". Položka menu "Print" je napojena na funkci zpětného volání *start_printing*, která spustí proces tisku. Zavoláním zmíněné funkce je vytvořena pomocí *gtk_print_operation_new()* nová tisková operace GtkPrintOperation. Následně jsou na GtkPrintOperation napojeny tři funkce zpětného volání (begin_print, draw_page a end_print), které budou volány jakmile dojde k zachycení signálů: *begin-print*, *draw-page* a *end-print* vyslaných na GtkPrintOperation. Samotná tisková operace se pak spustí zavoláním *gtk_print_operation_run()*, která zobrazí dialogové okno pro výběr tiskárny a nastavení dalších možností tisku. Jakmile je dokončena práce s dialogovým oknem jsou na GtkPrintOperation vyslány zmíněné signály.

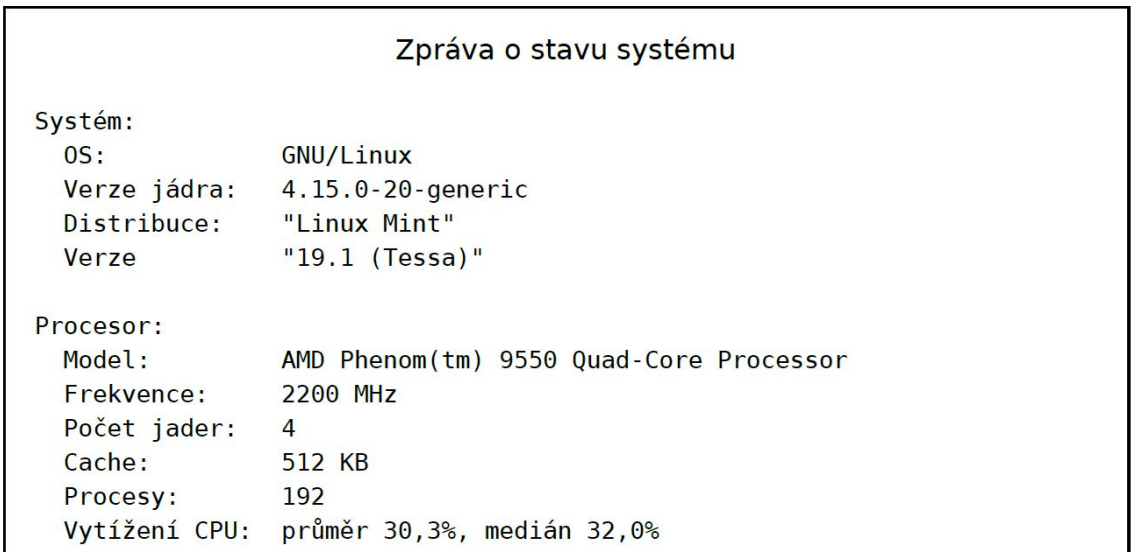
Před samotným tisknutím stránky je vyslán signál *begin-print*. Po zachycení tohoto signálu je zavolána funkce s podobným názvem begin_print, která připraví obsah pro tisk, nastaví velikost písma, vypočítá celkový a maximální možných počet řádků na stránku a celkový počet tisknutých stránek. Veškeré hodnoty jsou pak uloženy do struktury PrintData pro další zpracování. Následně je signál *draw-page* vysílán pro každou tisknutou stránku zvlášť. Po jeho zachycení se provede funkce s názvem draw_page, která se postará o vykreslení dané stránky. Před ukončením tiskové operace je ještě vyslán signál *end-print* na který reaguje funkce s názvem end_print, která jen uvolní tiskem alokovanou paměť.

### 3.7.1 Struktura reportu

Vytisknutý report má následující strukturu. Vytisknutá stránka bude mít v záhlaví nadpis "Zpráva o stavu systému". Na dalších řádcích už budou vytisknuté získané informace. Informace budou rozděleny a tisknuti do čtyř sekcí: *Systém*, *Procesor*, *RAM* a *Ostatní*.

V sekci *Systém* budou vytisknuty informace o operačním systému (typ, verze jádra, název distribuce a verze distribuce). V sekci *Procesor* budou zase vytisknuty hlavní parametry procesoru spolu s dopočítanou průměrnou hodnotou a mediánem jeho vytížení za sledované období. V této sekci bude také ještě uveden celkový počet spuštěných procesů v době tisknutí reportu. V sekci *RAM* je vytisknuta statistika využití operační

paměti. Je zde tedy uveden přehled jejího aktuálního stavu a dále ještě průměrná hodnota a medián využití paměti za sledované období. V poslední sekci s názvem *Ostatní* už je jen uveden model/označení detekované grafické karty a základní desky. Náhled na pouze část vytisknutého dokumentu se vzorovými hodnotami je zobrazen na obrázku 12. Náhled na strukturu celého tisknutého dokumentu je uveden v příloze 7.1.




**Obrázek 12: Struktura tisknutého reportu**

## 3.8 Vytvoření souboru makefile a kompilace programu

Aby bylo možné aplikaci spustit je nutné z konečných pěti zdrojových souborů sestavit jeden výsledný spustitelný program. Přesně k tomuto účelu poslouží program make. Nejprve je ale nutné sestavit soubor Makefile, do kterého jsou zapsány instrukce, jak se má výsledný spustitelný soubor vytvořit.

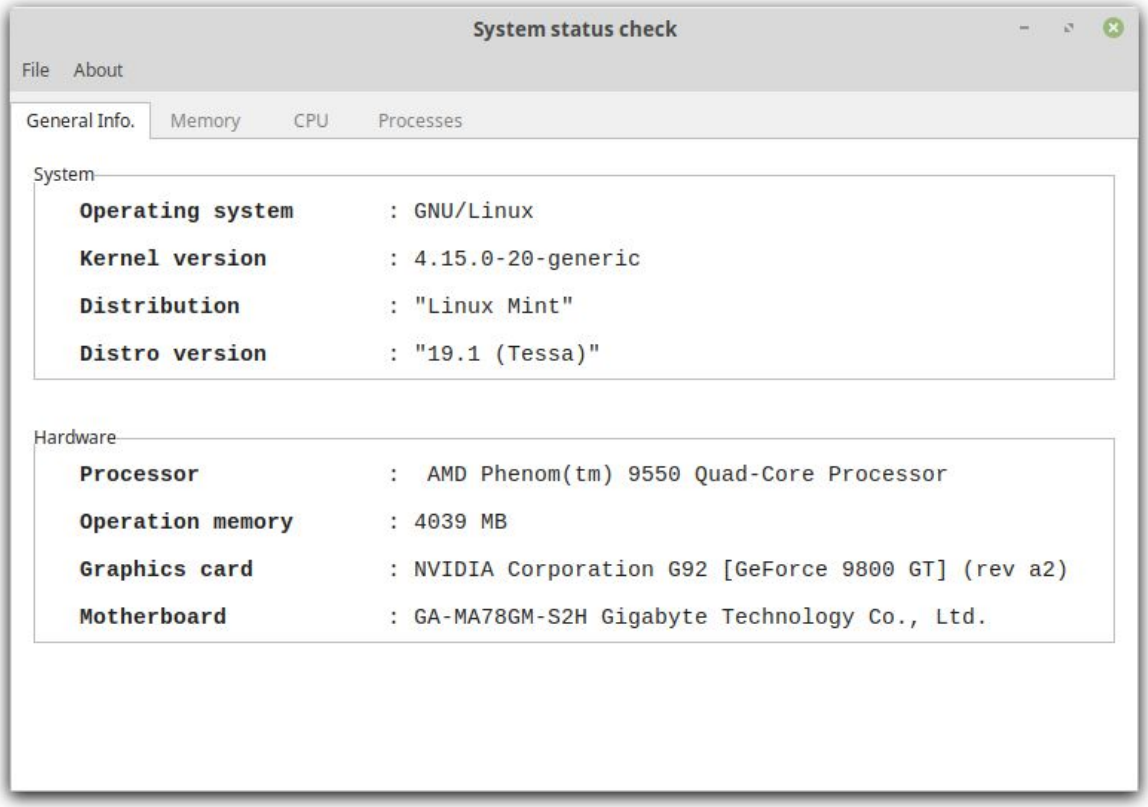
```
all: app
app: callback.c interface.c main.c data.c app_gtk.h
     gcc `pkg-config gtk+-3.0 --cflags` -g -Wall -pedantic
          -o status_check callback.c interface.c main.c data.c
          -lm `pkg-config gtk+-3.0 --libs`
clean:
     rm -f app
```

Jakmile je soubor Makefile uložen, stačí už jen spustit v terminálu příkaz make pro přeložení programu. Pro spuštění programu je nutné mít GTK+ alespoň ve verzi 3.16.

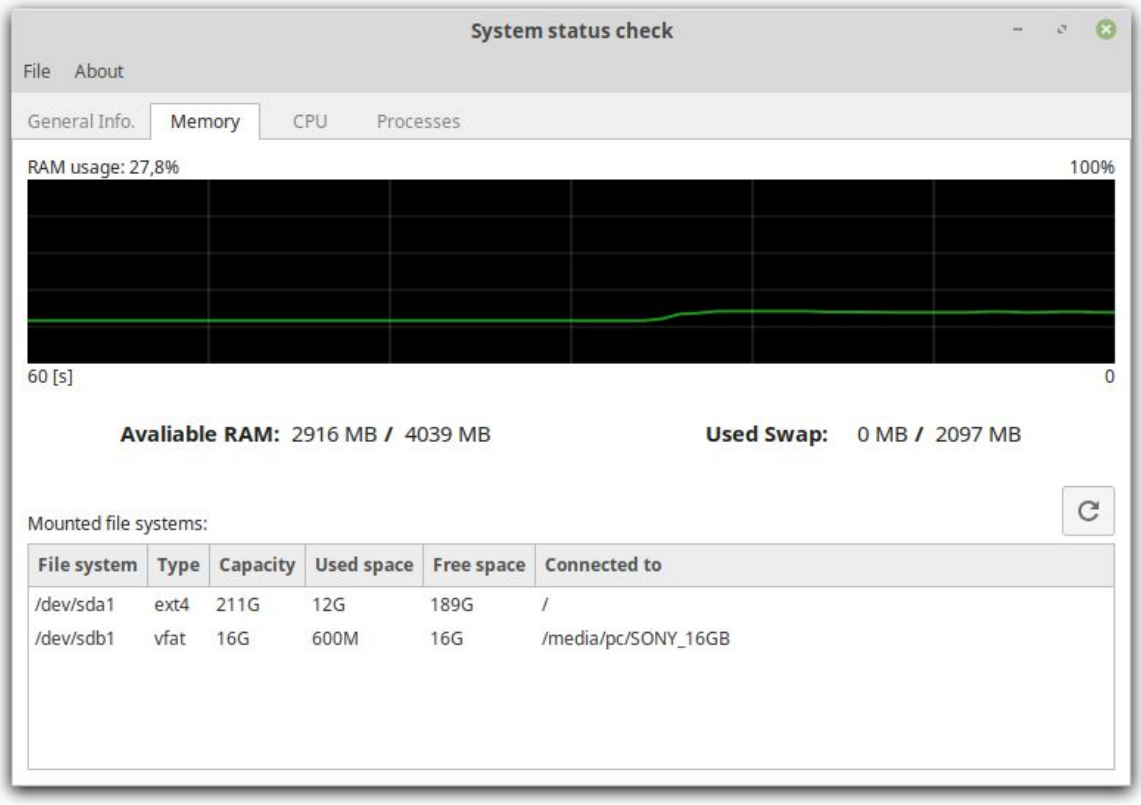
```
make -f Makefile
```

### 3.8.1 Spuštění programu

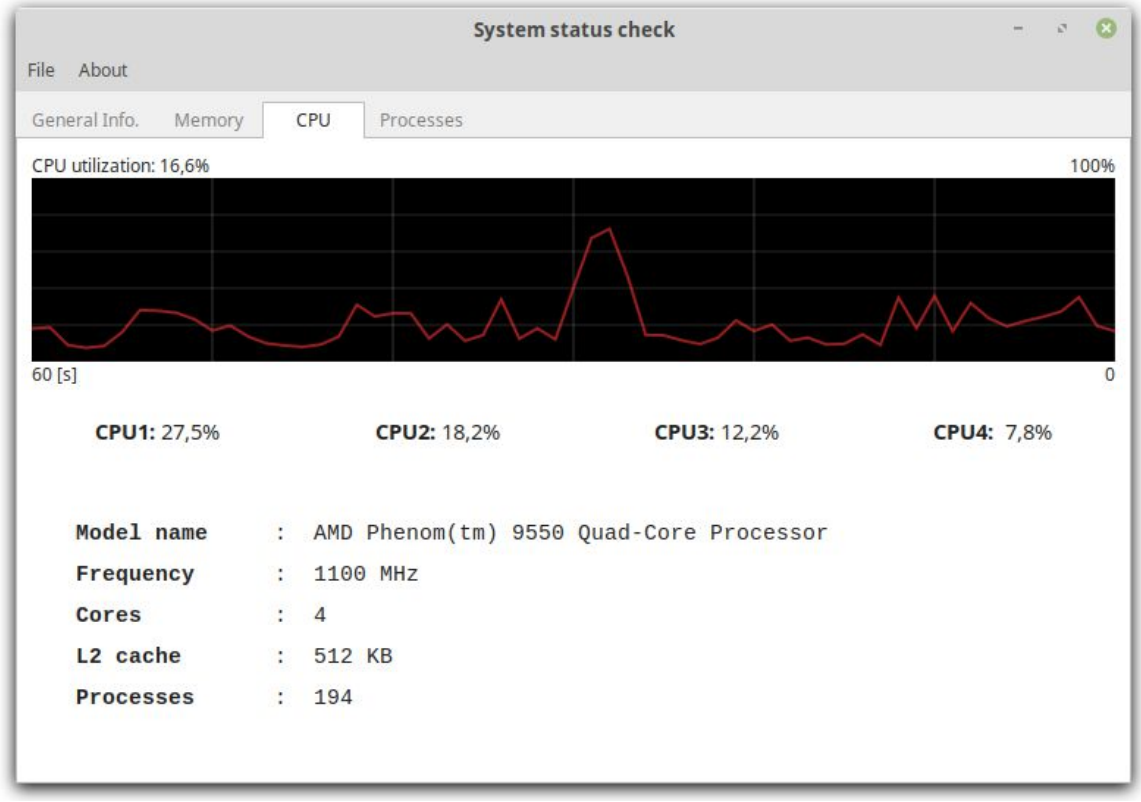
Výsledná aplikace lze pak spustit zadáním příkazu *./status_check* do příkazového řádku. Poté se zobrazí okno programu s aktivovanou první kartou, která zobrazuje základní parametry sledovaného systému (obrázek 13). Následně se lze mezi jednotlivými kartami přepínat. Výsledek je zobrazen na následujících obrázcích.



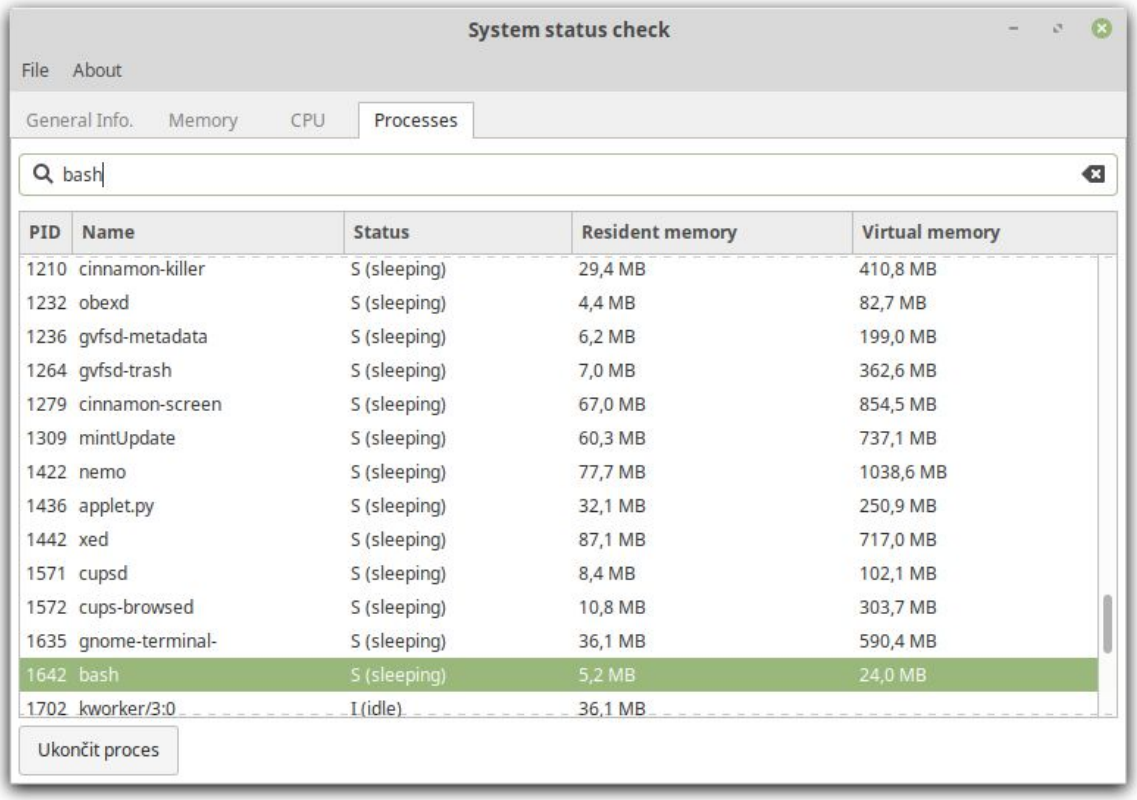

**Obrázek 13: Výstup - karta s obecnými informacemi**




**Obrázek 14: Výstup - karta s informacemi o paměti**




**Obrázek 15: Výstup - karta s informacemi o procesoru**




**Obrázek 16: Výstup - karta s informacemi o procesech**

Veškeré zdrojové kódy programu jsou nahrány na přiložené CD.

# 4 Výsledky a diskuze

Výsledkem této práce je aplikace s grafickým uživatelským rozhraním, která průběžně monitoruje stav systému. Aplikace je určená pro operační systém Linux a pomocí aplikace jsou průběžně sledovány základní informace o daném systému a vytížení/využití vybraných systémových zdrojů.

Pro linuxové systémy existují velmi dobré nástroje, které se zabývají sledováním často jen určitých oblastí systému. S většinou takovýchto nástrojů se pak pracuje skrze rozhraní příkazového řádku. Na jednu stranu to pro některé uživatele může být jednodušší způsob jak získávat informace o systému. Na druhou stranu to zase pro některé uživatele může být méně srozumitelný způsob jak získat požadované informace.

Aplikace vytvořená v rámci této práce má za úkol poskytnout základní přehled o stavu sledovaného systému s možností tento přehled poté vytisknout či uložit do souboru. Grafické rozhraní aplikace je navrženo tak, aby případné budoucí rozšíření aplikace o zobrazení dalšího okruhu informací bylo co nejjednodušší. To by bylo realizováno přidáním další karty do grafického rozhraní, která by poskytovala další informace např. informace o síťových zařízeních a síťovém provozu atd. V budoucnu by se i dala aplikace rozšířit o funkci tzv. logování, kdy by se automaticky zaznamenávalo např. kdy a jaký proces byl za běhu aplikace spuštěn či vypnut apod.

Aplikace byla vyvíjena na 64 bitové linuxové distribuci Linux Mint verze 19.1. Dále byla funkčnost aplikace otestována na linuxové distribuci Ubuntu 18.04.4 (64 bit), Kali Linux 2019.3 (64 bit), Fedora 31 Workstation a Linux Mint 19.3 (32 bit). Na zmíněných distribucích nebyly zjištěny žádné problémy. Hardwarové parametry počítače na kterém byla aplikace vyvíjena a testována jsou uvedeny na obrázcích z výstupu aplikace v kapitole 3.8.1 nebo v příloze 7.1.

# 5 Závěr

Cílem práce bylo vytvořit aplikaci pro operační systém Linux, která bude průběžně monitorovat aktuální stav systému. Dále umožňuje získané informace vytisknout případně uložit do souboru. Vytvořená aplikace disponuje grafickým uživatelským rozhraním. Pro psaní této práce byl použit programovací jazyk C.

V teoretické části práce byly uvedeny postupy, kterými lze aplikace pro linuxový operační systém sestavovat. Konkrétně jak vytvořit zdrojové soubory a následně je přeložit. Dále zde byly také popsány nástroje, které lze využívat při ladění aplikace. V poslední řadě zde byla uvedena knihovna GTK a jak pomocí této knihovny obohatit aplikaci o grafické uživatelské rozhraní.

V praktické části již byla vytvořena samotná aplikace na sledování stavu systému. Nejprve bylo analyzováno, co by měla aplikace umožňovat a jaké informace bude zpracovávat. Následně bylo přistoupeno k návrhu vzhledu aplikace. Byly navrženy jednotlivé části aplikace pomocí wireframů, pomocí kterých bylo naznačeno, kde by měly být umístěny jednotlivé ovládací prvky a jakým způsobem budou informace zobrazeny. Z těchto návrhů bylo následně vycházeno při vytváření grafického uživatelského rozhraní aplikace pomocí knihovny GTK+.

Při programování aplikace byl zdrojový kód rozdělen do pěti zdrojových souborů na základě povahy daného kódu. Tím bylo například docíleno, že byl oddělen kód pro vytvoření grafického rozhraní od kódu pro získávání dat ze systému. Nakonec byl pomocí nástroje *make* vytvořen ze zmíněných pěti zdrojových souborů jeden výsledný spustitelný soubor.

# 6 Seznam použitých zdrojů


1. Stones, Richard
   Linux : začínáme programovat [4.vyd.] / Richard Stones, Neil Matthew. Praha : Computer Press, 2008. 829 s. váz. ISBN:978-80-251-1933-4

2. Masters, Jon
   Linux profesionálně : programování aplikací / Jon Masters, Richard Blum. 1. vyd.. Brno : Zoner Press, 2008. 539 s. brož. ( Encyklopedie Zoner Press ) ISBN:978-80-86815-71-8

3. Mitchell, Mark
   Pokročilé programování v operačním systému Linux / Mark Mitchell, Jeffery Oldham, Alex Samuel. Praha : SoftPress, 2002. 320 s. brož. ISBN:80-86497-29-1

4. Kernighan, Brian W.
   Programovací jazyk C / Brian W. Kernighan, Dennis M. Ritchie. 1. vyd.. Brno : Computer Press, 2006. 286 s. brož. ISBN:80-251-0897-X

5. Dostál, Radim
   C/C++ : hotová řešení / Radim Dostál. 1. vyd.. Brno : Computer Press, 2009. 376 s. + CD brož.; CD disky ISBN:978-80-251-2190-0

6. Sobell, Mark G.
   Mistrovství v Linuxu : příkazový řádek, shell, programování / Mark G. Sobell. 1. vyd.. Brno : Computer Press, 2007. 878 s. váz. ISBN:978-80-251-1726-2

7. Herborth, Chris
   Unix a Linux : názorný průvodce programátora / Chris Herborth. 1. vyd.. Brno : Computer Press, 2006. 288 s. brož. ISBN:80-251-0978-X

8. Linux. *Linux* [online]. [cit. 2019-11-27]. Dostupné z: http://www.linux.cz

9. Distribuce Linuxu. *Linuxcesky* [online]. [cit. 2019-11-27]. Dostupné z: https://linuxcesky.cz/distribuce-linuxu/

10. Distribuce. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce

11. Distribuce: Ubuntu. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/ubuntu

12. Distribuce: Linux Mint. *Linu EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/linux-mint

13. Přehled linuxových distribucí. *ROOT* [online]. [cit. 2020-03-01]. Dostupné z: https://www.root.cz/texty/prehled-linuxovych-distribuci/

14. Distribuce: openSUSE. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/opensuse

15. Distribuce: Mandriva Linux. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/mandriva-linux

16. Distribuce: Knoppix. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/knoppix

17. Distribuce: CentOS. *Linux EXPRES* [online]. [cit. 2020-03-01]. Dostupné z: https://www.linuxexpres.cz/distribuce/centos

18. Gedit. *Gnome* [online]. [cit. 2020-02-16]. Dostupné z: https://wiki.gnome.org/Apps/Gedit

19. Programování pod Linuxem pro všechny (4). *ROOT* [online]. [cit. 2020-02-16]. Dostupné z: https://www.root.cz/clanky/programovani-pod-linuxem-knihovny-uvod/

20. Testování. *Sallyx* [online]. [cit. 2020-02-16]. Dostupné z: https://www.sallyx.org/sally/c/linux/testovani

21. Valgrind. *Valgrind* [online]. [cit. 2020-02-16]. Dostupné z: https://valgrind.org/

22. Valgrind's Tool Suite: Other Tools. *Valgrind* [online]. [cit. 2020-02-16]. Dostupné z: http://www.valgrind.org/info/tools.html#others

23. Valgrind User Manual: Helgrind: a thread error detector. *Valgrind* [online]. [cit. 2020-02-16]. Dostupné z: http://valgrind.org/docs/manual/hg-manual.html

24. Language bindings. *GTK* [online]. [cit. 2019-10-09]. Dostupné z: https://www.gtk.org/language-bindings.php

25. GTK. *GTK* [online]. [cit. 2019-10-09]. Dostupné z: https://www.gtk.org/

26. Referenční příručka k API. *GNOME Developer* [online]. [cit. 2020-03-02]. Dostupné z: https://developer.gnome.org/references.html.cs

27. Overview. *GTK* [online]. [cit. 2019-10-04]. Dostupné z: https://www.gtk.org/overview.php

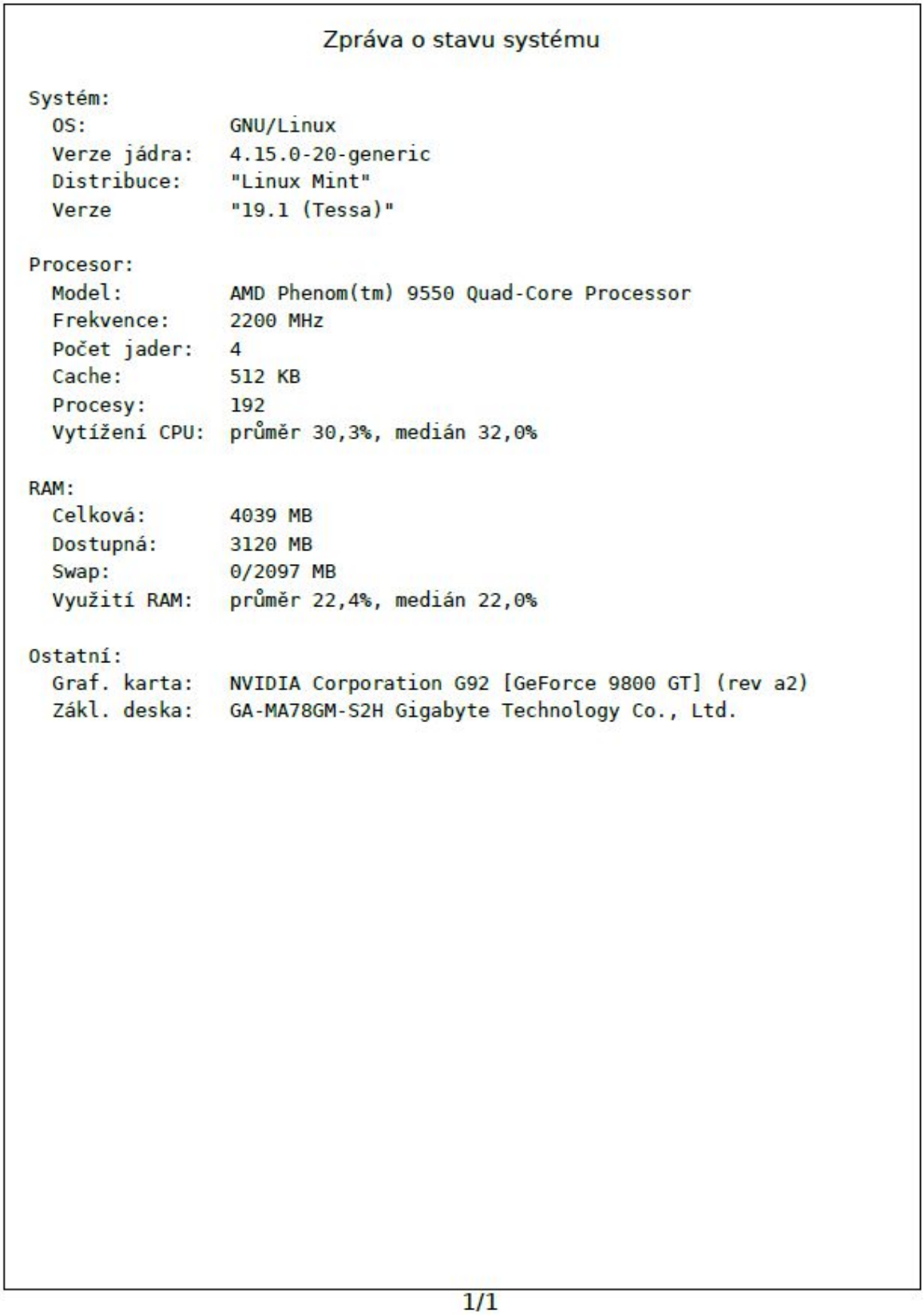
28. GtkListBox. *GNOME Developer* [online]. [cit. 2020-03-06]. Dostupné z: https://developer.gnome.org/gtk3/stable/GtkListBox.html

29. GtkBox: gtk_box_pack_start. *GNOME Developer* [online]. [cit. 2020-01-08]. Dostupné z: https://developer.gnome.org/gtk3/stable/GtkBox.html#gtk-box-pack-start

30. Gentoo: aneb vyplatí se stále kompilovat? *Linux EXPRES* [online]. [cit. 2020-01-10]. Dostupné z: https://www.linuxexpres.cz/distro/gentoo-aneb-vyplati-se-stale-kompilovat

# 7 Přílohy

## 7.1 Náhled na celou stránku tisknutého reportu



Zpráva o stavu systému

```
Systém:
  OS:             GNU/Linux
  Verze jádra:    4.15.0-20-generic
  Distribuce:     "Linux Mint"
  Verze           "19.1 (Tessa)"

Procesor:
  Model:          AMD Phenom(tm) 9550 Quad-Core Processor
  Frekvence:      2200 MHz
  Počet jader:    4
  Cache:          512 KB
  Procesy:        192
  Vytížení CPU:   průměr 30,3%, medián 32,0%

RAM:
  Celková:        4039 MB
  Dostupná:       3120 MB
  Swap:           0/2097 MB
  Využití RAM:    průměr 22,4%, medián 22,0%

Ostatní:
  Graf. karta:    NVIDIA Corporation G92 [GeForce 9800 GT] (rev a2)
  Zákl. deska:    GA-MA78GM-S2H Gigabyte Technology Co., Ltd.
```

1/1