ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Odlehčený systém správy procesů během spouštění a ukončení chodu operačního systému Linux

David Štorek

květen 2021
Vedoucí práce: Ing. Pavel Troller, CSc.

/ Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 20. 5. 2021

.....

Abstrakt / Abstract

Tato práce se zabývá popisem a některých omezení části inicializačního procesu Sinuxu založeného na operačním systému Linux a jejich možnými řešeními. Jedno z nich je pak v podobě programu Postinit je pak implementováno. Struktura a chování tohoto programu je také diskutována.

Klíčová slova: Linux; Sinux; init, SysV; bakalářská práce

This document deals with description of certain restrictions of part of inicialization process of Sinux based on Linux operating system and their possible solutions. One of those solutions is then implemented in a form of program Postinit. Description of structure a behavior of this program is also discussed.

Keywords: Linux; Sinux; init; SysV; bachelor thesis

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Tato práce se zabývá omezeními (a jejich řešeními) procesu daemons který v distribuci operačního systému linux [1] zvaného Sinux funguje jako nadstavba SysV-style [2–3] init procesu. Jsou zde popsány problémy se současným stavem, návrhy a analýza jejich možných řešení a detailní popis implementace a funkcionality vybraného řešení - programu Postinit. Na závěr jsou zde také popsány nedostatky nového řešení a jejich dopad na funkcionalitu systému.

1.1 Cíl práce

Cílem práce je najít nebo navrhnout a implementovat náhradu daemons scriptu sloužícího ke spouštní a ukončování procesů systému při změně runlevel [4] na operačním systému Sinux. Toto řešení by mělo nemělo příliš omezovat současnou funkcionalitu daemons scriptu a zároveň by mělo přidat funkce pro definice a zpracování závislostí mezi jednotlivými procesy.

1.2 Potřeba řešení

Systém Sinux se sice nevužívá přiliš hojně, ale zato se používá na velmi různých prostředích - od telefoních ústředen a serverů až po embeded systémy. Tato různorodost způsobuje že každá instalace systému provozuje odlišné procesy které je nutné spravovat init systémem. Ne každý z těchto různorodých procesů je ale možné spusit kdykoli a je nutné aby pro jejich správný běh byl při startu jiný proces už spuštění a inicializovaný. V současném stavu tohoto není možné docílit, neboť procesy se při změně runlevelu startují sekvenčně pouze podle jejich definované priority a ihned. Toto může způsobit start procesu i když prostředí ještě není plně připraveno pro jeho běh a proces zůstane v nefunkčním stavu.

Kapitola 2
Současný stav

Script daemons funguje jako rozšíření init [5] procesu které má za úkol spouštět a ukončovat většinu procesů běžících na systému na základě změny runlevel. Je implementovaný jako script pro bash [6] spouštěného přímo procesem init při každé změně runlevel.

2.1 Spouštění scriptu

Spuštění daemons scriptu je docílené konfigurací hlavního souboru procesů programu SysVinit [5] zvaného **inittab**. Tato konfigurace obsahuje následující část:

```
l0:0:wait:/etc/init.d/daemons update
l1:1:wait:/etc/init.d/daemons update
l2:2:wait:/etc/init.d/daemons update
l3:3:wait:/etc/init.d/daemons update
l4:4:wait:/etc/init.d/daemons update
l5:5:wait:/etc/init.d/daemons update
l6:6:wait:/etc/init.d/daemons update
l7:7:wait:/etc/init.d/daemons update
```

Konfigurace procesů o které se má script daemons starat je definována v souboru daemon.config.

2.2 Funkcionalita daemons

Script nabízí 3 módy běhu:

- **UPDATE** - nejčastěji používaný; Porovná současný a předchozí runlevel a vypne všechny procesy které byly přítomné v předchozím runlevelu ale už nejsou v tomto. Dále pak nastartuje všechny procesy které nebyly přítomné v minulém runlevelu ale jsou v tomto.
- **SUSPEND** - suspenduje (pozastaví) běh všech procesů které právě běží
- **RESUME** - obnový běh suspendovaných (pozastavených) proces;.

Script dále definuje 4 různé typy procesů:

- **D** - daemon - Spustí se definovaným příkazem a vypíná se „zabitím“
- **S** - script - Spuštění i vypínání probíhá definovaným příkazem s přidaným parametrem „start“ nebo „stop“
- **C** - command - Příkaz se jen spustí a nevypíná se
- **K** - kill - Příkaz se spustí pouze při vypnutí procesu

Chování jednotlivý typů procesů při běhu scriptu je definováno tabulkou 2.1.

akce	start	stop	suspend	resume
D	cmd	kill	kill	cmd
S	cmd start	cmd stop	cmd suspend	cmd resume
C	cmd	-	-	-
K	-	cmd	-	-

Tabulka 2.1. Chování procesů při běhu daemons scriptu.

2.2.1 Konfigurační soubor procesů daemon.config

Tento soubor definuje seznam procesů o které se má script daemons starat a jejich konfigurace.

Každý proces je zaznamenaný na novém řádku a jednotlivá pole jsou oddělena jedním nebo více bílými znaky.

Řádky začínající znakem '#' značí komentáře a řádky začínající znakem ';' jsou neplatné procesy. Oba typy řádků jsou scriptem ignorovány.

Příklad části souboru:

```
### Here go the daemons and services which should be
### initiated for EVERY runlevel.
12345 00 S root udev /etc/init.d/udev
# Hardware init first and shutdown last.
12345 01 S root hardware /etc/init.d/hardware
# Load kernel modules which don't load automatically.
12345 05 S root modules /etc/init.d/modules
# PCMCIA services should start soon,
# maybe we have disks connected over them.
;12345  07 S root pcmcia /etc/init.d/pcmcia
# System devices setup
12345 10 S root devboot /etc/init.d/devboot
# Check and mount root filesystem
12345 13 S root rootfs /etc/init.d/rootfs
# Check and mount local filesystems
12345 15 S root localfs /etc/init.d/localfs
```

Význam jednotlivých polí platných řádků v pořadí v jakém se vyskytují je popsán v následující části:

- **seznam runlevelů** - seznam runlevelů ve kterých má tento proces běžet
- **pořadí** - určuje pořadí startování procesu. Proces s nižším pořadím bude spuštěn dříve nebo ukončený později než proces s vyšším pořadím
- **typ procesu** - typ procesu - může být d,s,c nebo k.
- **uživatel** - jméno účtu uživatele systému pod kterým se proces bude spouštět
- **jméno procesu** - jméno spouštěného procesu. Používá se k identifikaci procesu na systému.
- **příkaz** - příkaz který bude použitý ke spuštění/ukončení/... procesu dle 2.1).

2.3 Implementace

Zde je popsaná implementace nejdůležitějších částí scriptu daemons. Tyto části definují hlavní chování procesu které bude nutné napodobit v novém řešení.

2.3.1 Pořadí spouštění procesů

Script určuje pořadí spuštění procesů podle následujícího pseudopříkazu:

```
<vyber procesy pro start> | sort | <spuštění/resume/suspend>
```

Pro ukončování procesů je situace podobná, avšak procesy se řadí v opačném pořadí:

```
<vyber procesy pro vypnutí> | sort -r | <ukončení>
```

2.3.2 Čtení a ukládání stavu procesu

Script se spouští při každé změně runlevelu a po vykonání všech akcí na definovaných procesech se ukončí. Je tedy nutné stav procesů uložit pro použití v dalším běhu scriptu.

Díky tomu že spouštění probíhá i ve velmi brzkých stadiích běhu operačního systému, většina filesystému je v režimu pouze pro čtení [3]. Výjimku tvoří složka /dev kde je možné vytvářet složky a symbolické odkazy. Toho script `daemons` využívá.

Zápis stavu znamená vytvoření symbolického odkazu na „stav“ procesu a probíhá příkazem:

```
ln -sf "<stav>" "/dev/init/<jméno procesu>"
```

Čtení stavu pobíhá analýzou textu vráceného příkazem:

```
ls -l "/dev/init/<jméno procesu>"
```

2.3.3 Spouštění procesů

Aby se procesy mohly spustit s uživatelem definovaným v konfiguračním souboru jako jejich vlastníkem, je nutné je spouštět pomocí nástroje **su** [7].

Pro procesy typu S se používá příkaz:

```
su <uživatel> -c "<příkaz> start"
```

Pro procesy typů D a C příkaz vypadá následovně:

```
su <uživatel> -c "<příkaz>"
```

Pro procesy type D pak script ještě hledá PID pod kterým byl proces spuštěn.

2.3.4 Ukončení procesů

Stejně jako při spouštění se i při vypínání využívá nástroje **su**.

Pro procesy typu S se používá příkaz

```
su <uživatel> -c "<příkaz> stop"
```

Pro procesy typu K příkaz vypadá následovně:

```
su <uživatel> -c "<příkaz>"
```

Odlišně je ale potřeba řešit procesy typu D. Ty nemají žádný standartní způsob ukončení a musí se „zabít“ pomocí nástroje **kill** [8]

```
kill -TERM <PID>
```

2.4 Popis problému

Hlavním nedostatkem současného systému je že pořadí spouštění a ukončování procesů závisí pouze na prioritě procesu definovaném v konfiguračním souboru procesů. To neumožňuje vytvářet závislosti mezi procesy a může vést k různým problémům:

- **chyba nastavení** - uživatel špatně určí prioritu procesu a ten se bude spouštět dříve než jiný proces na jehož běhu závisí.
- **chybný start procesu** - pokud některý z procesů při startu selže, tak se start všech následujících procesů zastaví. A to i těch které chybný proces pro svůj běh nevyžadují.
- **nesprávný běh procesu** - Některé procesy nemusí závist pouze na správném startu jiného procesu, ale i na jeho dalších stavech které se mohou měnit i po úspěšném dokončení jeho startu. Příkladem je třeba dostupnost sítě po spuštění programu **netconfig**. I když program proběhnul správně a vše s připojením je v pořádku, přístup do sítě není ihned možný. Načtení IP adresy a ustálení připojení může trvat několik sekund. Pokud se tedy proces spustí ještě než se tak stane, nemusí naběhnout korektně.

Kapitola 3
Možná řešení nedostatků

Finální řešení stávajících nedostatků bude integrované do existujících systémů založených na Sinuxu. Z toho důvodu je nutné aby nové řešení splňovalo několik podmínek a požadavků:

- **Zachování stávajicí funkcionlity** - Současný systém inicializace procesů je poněkud mnimalistický a věškerá funkcionalita kterou nabízí je využívána. Proto není možné žádnou z funkcí odebrat je nutné aby jejich chování zůstalo bez zásadních změn.
- **Malá paměťová stopa** - Sinux běží na mnoha typech zařízení od serverů po embeded systémy. Zejména kvůli druhé variantě kdy zařízení může mít i jen 32MB paměti RAM, je nutné aby nové řešení využívalo co nejméně systémových zdrojů.
- **Snadná integrace** - Nové řešení bude použité v mnoha stávajicích systémech a je kvůli jednoduchosti integrace je žádoucí aby jeho použití bylo co nejpodobnější stávající situaci.
- **možnost definice závislostí** - Přidání nové funkcionality která umožní nakonfigurovat závislosti mezi procesy a zajistí jejich správné inicializační pořadí.

Existují 3 základní způsoby řešení nedostatků stávajícího systému:

- Změna init procesu na jiný, který bude nativně podporovat nové požadavky
- Úprava stávajícího daemons scriptu.
- Vytvořit náhradu za stávající daemons script.

3.1 Změna init procesu

script daemons je specifický systému Sinux a jeho údržba a případný další vývoj je odkázaná na uživatele systému. Protže systém není příliš rozšířený a uživatelů není mnoho, údržba a vývoj scriptu nejsou příliš časté a stojí uživatele hodně času a úsilí. Použití již existujícího nástroje který je se ve světě Linuxu vyskytuje v mnohem větší míře znamená více lidí pro vývoj, a tedy i více funkcionality a nástrojů pro jeho použití. Použití existujícího řešení by tedy ušetřilo uživatelům Sinuxu čas strávený vývojem funkcionality která již jinde existuje.

Výběr některých existujících řešení a jejich vlastností:

3.1.1 Systemd

Jedná se o jednu z nejpoužívanějších kolekcí nástrojů a programů pro inicializaci [9] systému Linux.

3.1.2 Výhody:

- **Funkcionalita** - Velké množství různých schopností včetně správy na základě systémových událostí, řešení závislostí mezi procesy a paralelního jejich paralelního spouštění
- **Podpora** - Tento systém je stále aktivně vyvíjený a udržovaný i velkými komerčními společnostmi jako je RedHat. Má tedy velmi dobrou podporu a komunitu.

- **Je moderní** - První zveřejnění tohoto systému bylo v roce 2010 a byl vyvinut s ohledem na nové technologie.

3.1.3 Nevýhody:

- **Komplexita** -Je velmi komplexní a na rozdíl od současného inicializačního procesu operačního systému Sinux mu není snadné porozumět
- **Konfigurace** - Konfigurace je velmi odlišná od současné. Přechod na tento inicializační systém by vyžadoval značné úsilí na straně uživatelů Sinuxu.
- **Velikost** - Díky velkému množství schopností zabírá systém větší množství paměti [10] než současná kombinace SysVinit a daemons scriptu. To může být pro některé embedované systémy problém.

3.1.4 Upstart

Tento program umožňuje inicializovat a spravovat operační systém Linux na základně událostí. Stejně jako Systemd také podporuje paralelní spouštění a řešení závislostí mezi procesy [11]. Je ale o něco starší a již není dále vyvíjený.

3.1.5 Výhody:

- **Funkcionalita** - Množství různých schopností včetně správy na základě událostí, řešení závislostí mezi procesy a jejich paralelní spouštění
- **Podobný** SysVinit - Tento program byl vyvinut aby fungoval jako snadná náhrada inicializačního programu SysVinit. Jeho integrace do systému by tedy byla přiměřeně snadná.

3.1.6 Nevýhody:

- **Konfigurace** - Konfigurace je odlišná od té v konfiguračním souboru daemons scriptu daemon.config. Přechod na tento inicializační systém by vyžadoval větší změnu na všech zařízeních kde by byl použit.
- **Podpora** - Není již dále vyvíjený a poslední verze byla vydána v roce 2014 [11]

3.2 Úprava daemons scriptu

Stávající způsob inicializace v podobě daemons scriptu fungoval dlouho a je pro uživatele známý. Jeho vylepšení a nová verze by proto znamenala pouze rychlou a snadnou změnu ve všech zařízeních které ho používají.

3.2.1 Výhody:

- **Obeznámenost** - Všichni uživatelé Sinuxu tento script znají a umějí s ním pracovat
- **Velikost** - script samotný zabírá jen okolo 7KB. Jeho běh sice vyžaduje také spuštění programů bash a awk [12], nicméně celková paměťová stopa celého procesu je komunitou Sinuxu považována za přijatelnou.
- **Čitelnost** - script lze otevřít v libovolném textovém editoru a celý kód programu přečíst. To umožňuje uživatelům lépe porozumět implementované funkcionalitě. Také to umožňuje jeho snadné úpravy.

3.2.2 Nevýhody:

- **Výkon** - Interpretace scriptu pomocí programu bash není příliš efektivní. Při současné komplexitě scriptu to není problém, ale přidání další funkcionality by to zvláště na slabších zařízeních znamenalo znatelné zpomalení inicializačního procesu.
- **Složitost implementace změny** - script nebyl vyvinut s myšlenkou „budoucí krytí“ a přidání správy závislostí mezi procesy by znamenalo rozsáhlé změny které by script změnily k nepoznání. To je srovnatelné s vyvinutím úplně nového software.

3.3 Vývoj nového řešení

Vyvinutí nového software jako náhrady za stávající daemons script je časově nejnáročnější ale umožňuje nejvyšší míru přizpůsobení funkcionality k požadavkům.

3.3.1 Výhody:

- **Funkcionalita** - Funkcionalita nově vyvinutého software se odvíjí pouze od požadavků a časové náročnosti na jejich implementaci. Při návrhu úplně nového řešení se nemusí brát ohledy na již existující části řešení a implementaci funkcionality lze navrhnout přesně podle požadavků.
- **Efektivita** - Program lze navrhnout tak aby neměl příliš nepotřebné nebo nechtěné funkcionality navíc. To zlepší efektivitu výpočtu a paměťovou stopu oproti komplexnějším existujícím řešením.

3.3.2 Nevýhody:

- **Časová náročnost** - Vývoj nového software trvá výrazně déle než použití již existujícího řešení.
- **Odladění** - Testování software při vývoji často neodhalí všechny jeho nedostatky při reálném nasazení. Lze tedy očekávat že první verze programu bude použitelná pouze na některých testovacích zařízeních.

3.4 Shrnutí a výběr

Existující nástroje jako jsou Systemd nebo Upstart splňují všechny požadavky na funkcionalitu a díky jejich rozšíření mají i velmi dobrou podporu komunity Linuxu. Jejich struktura a konfigurace ale bohužel nebývá kompatibilní se současným řešením inicializačního procesu v Sinuxu. Také mívají mnoho funckionality navíc která by v nebyla využita a znamená tedy pouze větší zátěž na systémové zdroje bez znatelných benefitů.

Méně komplexní inicializační program Upstart tímto problémem sice netrpí, zato ale již není vyvíjený a jeho podpora je omezená.

Úprava stávajícího daemons scriptu by sice na první pohled zachovala zdání menší náročnosti než vývoj nového programu, ovšem díky jeho stávajícímu návrhu tomu tak není. Při úpravě by byl vývoj také limitovaný pouze na prostředí ve kterém script funguje nyní - zejména bash a program awk.

Poslední možnost je tedy vývoj nového software který bude implementovat funkcionalitu daemons scriptu a rozšíří ji o správu závislostí mezi jednotlivými řízenými procesy.

Díky tomu že vývoj není příliš závislý na existujících řešeních inicializačního procesu, existuje tu prostor pro návrh programu který bude výpočetně i paměťově efektivnější než stávající daemons script.

Řešením tedy bude návrh a implementace programu který bude nahrazovat současný daemons script a který umožní definovat a spravovat závislosti mezi procesy.

Kapitola 4
Návrh programu Postinit

Program je potřeba navrhnout tak aby se použitím podobal daemons scriptu. To platí i pro konfigurační soubor řízených procesů.

4.1 Návrh použití programu

4.1.1 Stanadartní spouštění

V současném stavu spouští init systém daemons script při každé změně runlevelu podle konfigurace v souboru inittab. Po dokončení řízení procesů definovaných v konfiguračním souboru procesů se script ukončí.

Toto chování Postinit zachová. Nebude se jednat o proces který stále běží na pozadí systému, ale bude se spouštět jen při změně runlevelu definovaném v inittab souboru a po dokončení řízení procesů se ukončí. Výchozí mód spuštění bude „update“ který provede start a vypnutí procesů podle definice runlevelu v konfiguračním souboru procesů.

Výchozí hodnoty runlevelů bude init předávat stejně jako v současném stavu - pomocí proměnných prostředí zvaných **RUNLEVEL** sloužící k uložení současného runlevelu a **PREVLEVEL** ukládající minulý runlevel.

4.1.2 Manuální spuštění

Program Postinit bude možné spustit také manuálně. Funkce dostupné uživateli budou

- **Pomoc** - vypíše příklad použití programu a možné konfigurace
- **Update** - provede update jako kdyby se program spouštěl pomocí init
- **Resume & Suspend** - umožní suspendovat běžící procesy a znovu spustit suspendované procesy
- **Kontrola nastavení** - Provede kontrolu konzistence nastavení procesů. Bude kontrolovat syntaxi konfiguračního souboru, a konzistenci procesů. To bude obsahovat kontrolu syntaktických chyb v konfiguraci, kontrolu existence cyklů závislostí a závislosti na procesech které nemohou běžet (například kvůli špatně nastavenému seznamu runlevelů).

4.1.3 Konfigurace procesů

Konfigurační soubor procesů programu Postinit bude kvůli jednoduchosti přechodu ze starého systému na nový vypadat velmi podobně jako stávající konfigurační soubor daemon.config.

Každý řádek bude buď komentář nebo konfigurace procesu.

Řádek komentáře bude začínat znakem „#“.

Řádek obsahující konfiguraci procesu pak bude obsahovat téměř stejná pole jako řádek konfigurace procesu v daemon.config oddělené libovolným počtem bílých znaků. Lišit se mezi sebou budou tak že konfigurace procesů programu Postinit nebude obsahovat pole priority, ale místo to bude obsahovat pole se seznamem závislostí které daný proces potřebuje pro svůj správný běh.

4.2 Typ procesu WAIT FOR

Některé procesy mohou pro svůj start vyžadovat nejen úspěšný start jiného procesu, ale i jeho inicializaci která může běžet i po ukončení startovacího procesu. Příkladem procesu který nemusí být plně inicializován ihned po startu je netconfig, kdy i za předpokladu že je vše správně nakonfigurované může získání IP adresy z DHCP serveru sítě trvat i několik sekund.

Některé procesy tedy vyžadují mechanismus který zajistí že se nebudou startovat dříve než bude splněna určitá podmínka. Toto bude řešeno pomocí nového typu procesu který program Postinit přidá mezi stávající typy procesů 2.1 zvaného WAIT FOR.

4.2.1 Chování

Proces typu WAIT FOR bude skript nebo program který zkontroluje zda je určitá podmínka splněna. Pokud ano, navrátí hodnotu OK. Pokud podmínka splněna nebude ale je pravděpodobné že časem splněna bude, vrátí hodnotu WAIT. Pokud bude zřejmé že podmínka splněna nebude nebo nastane chyba, vrátí hodnotu ERROR.

Program Postinit při startu tohoto typu procesu zkontroluje návratovou hodnotu a podle výsledku provede akci:

- **OK** - pokračuj ve startování všech procesů
- **WAIT** - pokračuj ve startu procesů které nezávisí na tomto procesu. Pokud ke startu zbývají pouze WAIT FOR procesy, chvíli počkej a proveď kontrolu znovu.
- **ERROR** - Zastav spouštění procesů závislých na tomto procesu a pokračuj ve startu ostatních nezávislých procesů.

4.3 Návrh běhu programu

Hlavní funkce programu - změna běžících procesů v závislosti na změně runlevel - je vyjádřena v grafu na obrázku 4.1.

4.3.1 Průběh startu procesů

Startování procesů musí brát v ohled závislosti mezi procesy. Program start začíná načtením současný stavu procesů a poté nastartuje vše co lze s ohledem na runlevel a již běžící procesy nastartovat. Poté si znovu načte stav procesů a znovu nastartuje vše co bude možné. Cyklus pokračuje dokud je možné nějaký proces nastartovat a také neexistuje proces který by čekal na start procesu typu WAIT FOR.

Celý algoritmus pro start procesů je znázorněný v grafu na obrázku 4.2.

4.3.2 Průběh vypínání procesů

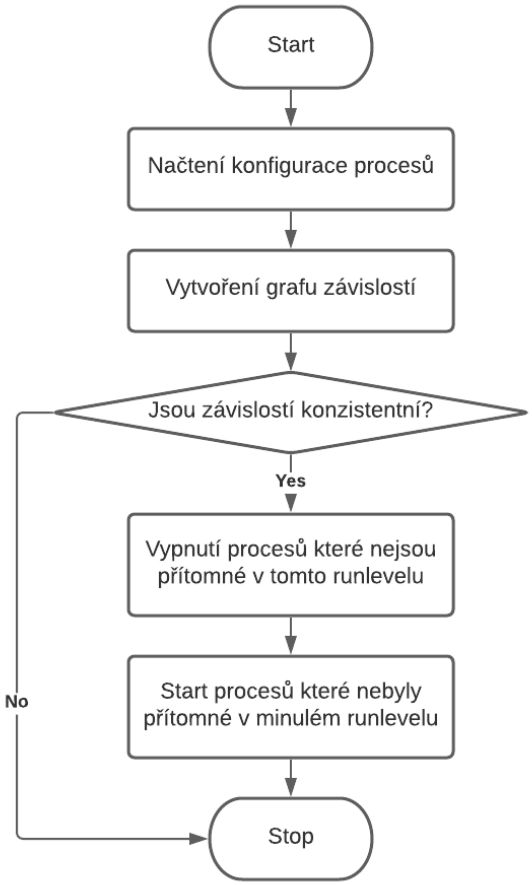
Vypínání procesů je založené na stejném principu jako jejich start, jen se volají funkce pro ukončení procesu místo funkcí pro jeho start. Procesy typu WAIT nebudou kontrolovány, takže odpadá potřeba čekat na splnění podmínky kterou kontrolují.

Průběh vypínání procesů je znázorněn v grafu na obrázku 4.3.

4.3.3 Průběh resume a suspend

Funkce resume odpovídá svým průběhem funkci start, jen se místo funkce „start process“ bude volat funkce „resume process“.

Suspendování procesu probíhá stejně jako vypínání procesů, ale opět s odlišnou funkcí konkrétní akce.



Obrázek 4.1. Flowchart hlavní funkce programu Postinit - správy procesů při změně runlevelu.

4.4 Výběr jazyku

Výběr jazyku ve kterém se bude program vyvíjet je značně omezený požadavkem na co nejmenší paměťovou stopu. Jakýkoli interpretovaný jazyk potřebuje pro svůj běh mít také nahraný interpreter, který bude zabírat místo v paměti RAM navíc. Výjimku tvoří bashscript interpretovaný pomocí `bash` který bude v systému již nahraný.

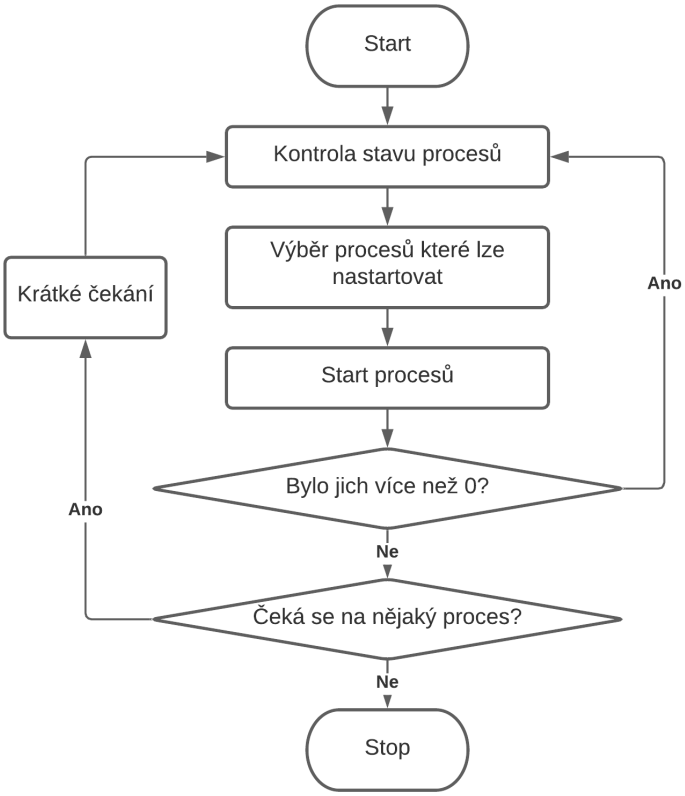
Z kompilovaných jazyků se standardně pro vývoj aplikací v `linuxu` používají `C` a `C++`.

4.4.1 Výhody bash:

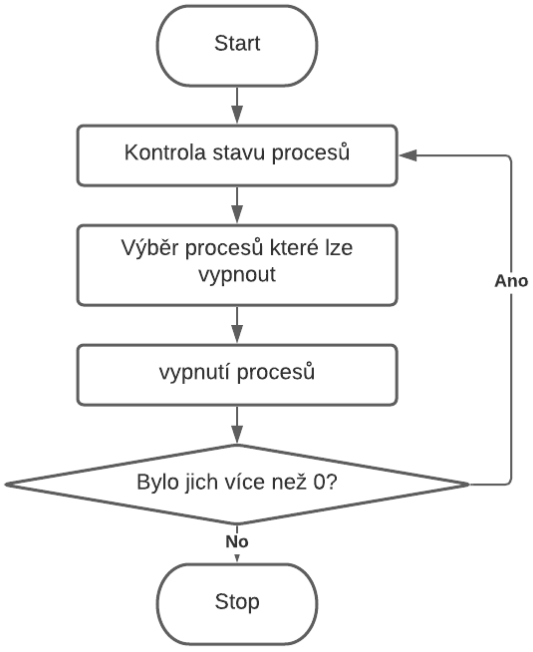
- **Čitelnost a snadné úpravy** - program napsaný v `bashcriptu` bude lidsky čitelný a snadno upravitelný i po nasazení.

4.4.2 Nevýhody bash:

- **nevhodný pro větší projekty** - Bashscript nenabízí žádné výchozí nástroje pro tvorbu komplexnějších datových struktur a vazeb mezi nimi. Díky povaze tohoto jazyku také programy s větším množstvím kódu často ztrácejí čitelnost.
- **Nároky na systémové prostředky** - Bashscript se interpretuje za běhu a tudíž je méně výkonný než C nebo C++. Toto může zvlášť u komplexnějších algoritmů znatelně



Obrázek 4.2. Flowchart průběhu startování procesů.



Obrázek 4.3. Flowchart průběhu vypínání procesů.

zpomalit průběh. Interpreter také spotřebovává více paměti než nativně běžící program.

4.4.3 Výhody C++:

- **Množství nástrojů** - C++ je velmi vyspělý jazyk který obsahuje ve své standardní knihovně obsahuje velké množství již implementovaných algoritmů a datových struktur. To velmi usnadní a urychlí vývoj.
- **Výkon** - Optimalizovaný program napsaný v C++ bude velmi rychlý v porovnání s bash.

4.4.4 Nevýhody C++:

- **Potřeba nahrát libstdc++** - Pokud chceme využít standardní knihovny v našem programu, musíme knihovnu při spuštění nahrát. Knihovna libstdc++ ale bohužel zabírá více než 1MB [13] a to není v souladu s naší podmínkou na minimální paměťovou stopu.

4.4.5 Výhody C:

- **Minimální paměťová stopa** - Velikost výsledného programu a jeho paměťová stopa v RAM za běhu a bude úměrná pouze množství funkcionality a nebude příliš záviset na velikosti standardní knihovny.
- **Výkon** - Optimalizovaný program napsaný v C bude velmi rychlý - srovnatelný s C++.
- **Integrace** - jazyk C nabízí množství nástrojů které umožňují velmi snadnou integraci do operačních systémů založených na Linuxu.

4.4.6 Nevýhody C:

- **Malá standardní knihovna** - Standardní knihovna neobsahuje žádné komplexní datové struktury ani algoritmy. Téměř vše musí být vyvinuto od začátku, nebo se musí využít kód třetí strany.

4.4.7 Výběr

Z předchozího výběru programovacích jazyků pro vývoj Postinitu nejméně vhodný bashskript. Výsledný software by sice byl snadno upravovatelný i po nasazení, nicméně struktura jazyka není příliš vhodná pro vývoj komplexnějších systémů.

Druhá možná volba - jazyk C++ - by byl ve své nejstandardnější podobě pro samotný vývoj nejvhodnější, bohužel by to ale znamenalo že výsledný software by pro svůj běh vyžadoval mít nahranou standardní knihovnu libstdc++ která by vyžadovala další systémové zdroje. Pokud by knihovna použita nebyla, jazyk by ztratil většinu své funkcionality a již by pro vývoj nebyl tak lákavý.

Poslední možnost v podobě jazyka C je pro vývoj programu Postinit tedy nejvhodnější. Sice nemá dostupných tolik nástrojů jako C++ a některé základní datové struktury budu muset být implementované při vývoji, ale zato nabízí snadnou komunikaci se systémem Linux a přitom výsledný program nebude zabírat příliš místa.

4.5 Nastavení vývojového prostředí

Samotný vývoj může sice probíhat na jakémkoli systému, nicméně celý proces bude přímočařejší pokud bude prostředí vývoje podobné jako cílový systém Sinux na kterém program poběží. Nebude tak potřeba řešit komplexitu spojenou „crosscompilováním“ (kompilováním kódu programu pro jiný systém než na jakém kompilace probíhá) a také se velmi usnadní implementace integračních testů.

Nejhodnější systém pro vývoj je tedy operační systému Linux. V tomto projektu byla využita jeho distribuce Ubuntu [14] s využitím IDE Netbeans 12.3 [15].

Ke spuštění kompilace a dalších kroků potřebných k vytvoření finální aplikace je využit nástroj make [16]. Tento nástroj vyžaduje vstupní konfigurační soubor zvaný makefile který definuje kroky prováděné při vytváření aplikace. V tomto případě je tento soubor spravovaný interním systémem IDE Netbeans 12.3 a není možné upravovat ručně.

Kapitola 5
Implementace Postinit

Při vývoji bylo potřeba řešit mnoho dílčích problémů jejichž řešení je podtstanou částí výsledného programu Postinit. Některé z nich jsou zde budou popsány a použité řešení je nastíněno v podobě pseudokódu.

5.1 Vector

Jedna ze základních datových struktur používaných ve vývoji téměř jakéhokoli software je pole prvků. Jedná se o lineární část paměti o fixní velikosti. Pokud ale při vytváření není známý počet prvků který chceme v poli uložit, vzniká problém.

Nejjednodušší řešení je vytvořil pole o velikost větší než jaký je maximální vyžadovaný počet prvků. To ale vede ke značné neefektivitě kdy pole bude často mnohem větší než kolik se ho skutečně používá.

U druhého možného řešení tento problém nevzniká - Na začátku se vytvoří pole menší velikosti. Pokud poté bude potřeba uložit více prvků než kolik se do pole vejde, vytvoří se pole nové s větší velikostí a prvky ze starého pole do něj překopírují.

Struktura používající tuto možnost se obvykle v jazyce C nazývá vector. Díky užitečnosti této struktury existuje mnoho různých jejích implementací. Program Postinit využívá opensource (licence MIT) projekt goldsborogh/vector [17].

5.1.1 Použití

Ukázka základního použití

```
//přidání hlavičkového souboru definujícího možné operace vektoru
#include "vector.h"

//vytvoření struktury pro uložení informací o vektoru
Vector vector;

//Samotné vytvoření vektoru pro ukládání Int o základní velikost 10 prvků
vector_setup(&vector, 10, sizeof(int));

//vložení prvku na konec
int x = 5;
vector_push_back(&vector, &x);

//načtení prvku na pozici 0 (první prvek)
int y = *(int*)vector_get(&vector, 0);

//smazání pole
vector_destroy(&vector);
```

5.2 Graf závislostí procesů

Graf závislostí je v Postinitu reprezentován polem uzlů grafu. Každý uzel ukládá pouze odkaz na proces ke kterému náleží a seznam uzlů na kterých závisí. Odkazy na tyto uzly mají formu jejich pozice v poli uzlů.

```
struct Uzel {
    Proces* proces;
    Index[] poziceZávislostí;
}
```

5.2.1 Stavba grafu závislostí

```
Graf vytvořGrafZávislostí(procesy){

    Graf graf
    graf.uzly = Uzel[procesy.size]

    for(proces in procesy){
        graf.uzly[indexof(proces)] = Uzel(proces)

        for(zavisiNa from proces.zavislosti){
            graf.uzly[indexof(proces)]
                .pridejZavislost(procesy.poziceOf(zavisiNa))
        }
    }

    return graf
}
```

5.3 Kontrola cyklů závislostí

Kontrola cyklů mezi závislostmi procesů je prováděna pomocí algoritmu využívajicí Deapth First Search (DFS) [18].

Hlavní funkce hledání cyklu:

```
bool obsahujeCyklus(graf){

    for(uzel in graf){
        if(DFSCycleSearch(uzel) == true){
            return true
        }
    }

    return false
}
```

Pomocná funkce hledání cyklu:

```
bool DFSCycleSearch(uzel){

    if (uzel.visited == false) {
        uzel.visited = true
        uzel.stack = true

         for(zavislost in uzel.zavislosti){
            if((zavislost.visited == false and DFSCycleSearch(zavislost))
                or zavislost.stack == true){
                return true
            }
         }
    }

    uzel.stack = false
    return false
}
```

5.4 Start procesů

Implementace startu procesů svou hlavní funkcionalitou odpovídá flowchartu startu procesů 4.2. Scripty typu WAIT FOR se chovají odlišně a je třeba je zpracovávat odděleně. Jejich běh může trvat nezanedbatelný čas a spouštět je při každém pokusu o start startovatelných procesů by znamenalo zpomalení programu. Start je proto vhodné odložit až na dobu když byly spuštěny všechny ostatní procesy které mohly být spuštěny.

```
bool shouldBeStarted(proces){
    return (dependenciesAreRunning(proces) and checkRunlevel(proces))
}

void startAllThatCanBeStarted(procesy, bool runWaitFor,
    *processesExecutedNr, *processesWaitingNr){

    processesExecutedNr = 0
    processesWaitingNr = 0


    if(runWaitFor == false){
        procesy = removeWaitForScripts(procesy)
    }

   started = 1

    while(started > 0){
        for(proces in procesy){
            if(shouldBeStarted(proces)){
                result = start(proces)

                if (process.type != WAIT_FOR) {
                    *processesExecutedNr += 1;
```

```
                }

                if(result == ERROR){
                    disableDependentProcesses(proces)
                }else if(result == RUNNING){
                    started++
                }
            }
        }

        if(nothingWasStarted){
            keepRunning = false;
        }
    }

    processesWaitingNr = countWaitingProcesses(procesy)
}

void startProcesses(procesy){
    bool processWaiting = 1

    while(processWaitingForStart){

        processesStartedNoWaitFor = 0;
        processesWaitingForWaitForScripts = 0;

        startAllThatCanBeStarted(procesy, false,
            &processesStartedNoWaitFor,
            &processesWaitingForWaitForScripts)

        startAllThatCanBeStarted(procesy, true,
            &processesStartedNoWaitFor, &processesWaiting)

        if(processesWaiting > 0){
            sleep
        }
    }
}
```

5.5 Vypnutí procesů

Vypínání procesů probíhá v opačném pořadí než jejich spouštění. Implementace

```
void stopProcesses(procesy){
    processesStopped = 1;
    while (processesStopped > 0) {
        for(proces in procesy){
            if(shouldBeStopped(proces)){

                bool allDependenciesStopped = true
                for(dependecy in listProcesesDependendOn(proces)){
```

```
                    if(dependecy.state == RUNNING){
                        allDependenciesStopped = false
                    }
                }

                if(allDependenciesStopped== true ){
                    stopProcess(proces)
                }
            }
        }
    }
}
```

5.6 Systémová volání

Program Postinit při svém běhu využívá řadu systémových funkcí Linuxu. Zejména se jedná o startování/vypínaní procesů a ukládání/načítání jejich stavu.

5.6.1 Procesu typu S

Proces typu S (Script) se zapíná i vypíná příkazem definovaným v konfiguračním souboru procesů s přidaným parametrem start/stop/suspend/resume.

Start scriptu:

```
#include <stdlib.h>
int resultCode = system("su %user -c %command start");
```

Vypnutí scriptu:

```
int resultCode = system("su %user -c %command stop");
```

5.6.2 Procesu typu D

Proces typue D (daemon) se spouští příkazem definovaným konfigurací procesu a vypíná se podle jeho PID získaného při spuštění

Start daemonu:

```
pid[] readProcessPids(prcoessName){
    FILE cmdResult = popen("pidof %processName", "r");
    char* result = fgets(line, BUFFER_SIZE_SYSTEM_READ, cmdResult);
    return parsePidsFromLine(result);
}


pid[] pidsBeforeStart =  readProcessPids(processName);
int resultCode = system("su %user -c %command start");
pid[] pidsAfterStart =  readProcessPids(processName);

//najde pid které je v pidsAfterStart a není v pidsBeforeStart
pid processPid = difference(pidsAfterStart, pidsBeforeStart);
```

Zastavení daemonu:

```
//Zastaví celou skupinu navázaných procesů.
result = system("kill -TERM -%pid);
if(result == ERROR){
    //Zastaví pouze 1 process podle PID
    system("kill -TERM %pid);
}
```

5.6.3 Procesy typů C, K a W

Tyto procesy se všechny startují stejným způsobem a nikdy se nevypínají. Po konci jejich spuštění je lze považovat za ukončené.

Spuštění:

```
int resultCode = system("su $user -c $command");
```

5.6.4 Persistence stavu procesů

Při každém spuštění programu se kontroluje současný stav procesů. Získávání těchto informací ze současného stavu systému by bylo výpočetně náročné a nespolehlivé. Je proto nutné stav procesů mezi jednotlivými běhy programu zachovat.

Program Postinit se spouští i velmi brzo po startu systému kdy většina filesystému ještě není připravena ke čtení a zapisování dat. I v tomto případě je ale nutné stav procesů ukládat. Te je umožněno díky tomu že je i v tomto stavu možné vytvářet symbolické odkazy ve složce /dev.

Zapsání stavu procesu:

```
unlink("/dev/postinit/%processName");
symlink("%processState %PIDorERRCode", "/dev/postinit/%processName");
```

Načtění stavu procesu:

```
link = readlink("/dev/init/%processName");
state = parseLink(link);
```

Kapitola 6
Použití Postinit

Program Postinit je konfigurovatelný pomocí parametrů při spuštění i pomocí konfiguračních souborů. Všechny parametry i možnosti spuštění programu jsou zde popsány.

Většina parametrů se dá nastavit jak pomocí parametrů příkazu, tak i pomocí konfiguračního souboru. Pořadí zdrojů načítání prametrů je následující (od shora dolů):

- Výchozí hodnota
- Proměnné prostředí (pouze runlevel a prevlevel)
- konfigurační soubor
- parametry zadané při spuštění

6.1 Parametry programu

Při spuštění programu bez parametrů, s chybným parametrem nebo pokud se některý z parametrů rovná „-help“ nebo „–help“, vypíše se nápověda použití programu se seznamem možných parametrů a jejich hodnot.

Jednotlivé parametry a jejich hodnoty jsou odděleny mezerou.

Možné parametry:

- **-c %, -config %** - Cesta k souboru s konfigurací
- **-m %, -mode %** - Zvolí mód aplikace. Validní hodnoty: NORMAL, N (výchozí) - spustí aplikace normálně; TEST_PROCESS, T - Pouze zkontroluje syntax a konzistenci konfiguračních souborů procesů. Nespouští ani nevypíná žádný proces.
- **-u %, -updateType %** - Určuje co udělat Validní hodnoty: UPDATE (výchozí) - Spustí a vypne procesy podle runlevel; SUSPEND - suspenduje běžící procesy, RESUME - znovu spustí suspendované procesy.
- **-v %, -verbosity %** - volba výřečnosti aplikace. Validní hodnoty: BASIC (výchozí) - normální výpis; VERBOSE - detailný výpis; SILENT - vypíše pouze chybová hlášení
- **-p %, -processesFile %** - Cesta k souboru s konfigurací procesů
- **-l %, -processesList %** - Cesta k souboru se seznamem dodatečných konfiguračních souborů prcoesů
- **-s %, -statusesDir %** - Cesta do adresáře který má být použit pro ukládání stavu procesů. Výchozí hodnota cesta je /dev/postinit
- **-t %, processCheckTimeout ¡¿** - Čas v sekundách po který se má čekat mezi testy skriptů typu WAIT FOR
- **-runlevel %** - Manuálně definuje současnou hodnotu runlevelu
- **-prevlevel %** - Manuálně definuje současnou hodnotu předchozího runlevelu

Příklady spuštění programu s parametry:

```
postinit -verbosity VERBOSE -c /etc/init.d/postinit.config
postinit -u SUSPEND -p /etc/init.d/postinit.processes
```

6.2 Konfigurační soubor

Nastavení konfigurovatelné v konfiguračním souboru je stejné jako nastavení konfigurovatelné pomocí parametrů.

Každý řádek konfiguračního souboru může být buď komentář (první nebílý znak je „#“) nebo parametr ve formátu jméno=hodnota

Příklad konfiguračního souboru:

```
#Tohle je komentář
verbosity=VERBOSE
processesFile=/etc/init.d/postinit.processes
processesList=/etc/init.d/postinit.installed
statusesDir=/dev/postinit/
#Hodnota je čas v sekundách
processCheckTimeout=10
```

6.3 Konfigurační soubor procesů

V tomto souboru jsou specifikovány všechny procesy spravované programem Postinit.

6.3.1 Struktura souboru

Každý řádek souboru kdy první nebílý znak je „#“ je komentář a je ignorován. Ostatní řádky jsou záznamy procesů. Jednotlivá pole informací o procesu jsou oddělena bílými znaky.

Pořadí a popis polí konfigurace procesu:

- **seznam runlevelů** - Seznam runlevelů na kterých má proces běžet
- **typ procesu** - 1 znak s typem procesu. Může být S, D, C, K nebo W (na velikosti nezáleží)
- **jméno procesu** - Jméno procesu. Také se používá k jeho hledání mezi běžícími procesy systému.
- **seznam závislostí** - Seznam jmen procesů oddělených čárkou na který tento proces závisí. Znak „
- “ je použit jako placeholder pro „Žádné závislosti“
- **uživatel** - Jméno uživatele který má být použit jako vlastník spouštěného procesu
- **příkaz** - Příkaz který se má spustit

6.3.2 Příklad konfigurace

```
2345    s       test    *       root    /etc/init.d/test
12345   d       test2   test1   root    /etc/init.d/test2
```

6.4 Integrace s SysVinit

Nejčastěji se program spouští při změně runlevelu. Je proto nutné ho přidat do konfiguračního souboru inittab:

```
l00:0:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l01:1:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l02:2:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l03:3:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
```

```
l04:4:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l05:5:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l06:6:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
l07:7:wait:/etc/init.d/postinit -u UPDATE -c /etc/init.d/postinit.config
```

Kapitola 7
Závěr

Tato práce se zabývala rozborem stávající nadstavby inicializačního systému init, jejími nedostatky, možnými řešeními těchto nedostatků, výběrem jednoho z nich a jeho implementací. Výsledkem práce je program Postinit který nahrazuje script daemons a jehož cílem je spouštění a vypínání procesů podle změn v runlevelu systému. Program také umožňuje definovat závislosti mezi jednotlivými procesy, které mají zajistit jejich správné pořadí startování a vypínání. Díky tomu že použití programu je velmi podobné původnímu řešení, je jeho integrace do stávajících zařízení poměrně jednoduchá a bez větších rizik.

7.1 Možnosti dalšího vývoje

Program Postinint sice nahrazuje funkcionalitu scriptu daemons při správě procesů, ale při testování a užívání programu byly objeveny situace ve kterých by bylo vhodné mít v programu další funkcionalitu navíc. jedná se například o:

- **Interaktivní mód** - Nyní program postinit pouze do konzole vypisuje co se děje, ale průběh není možné nijak ovlivnit. Přidání interaktivního módu kdy se program při každé změně stavu procesu aktivně táže uživatele zda se má proces opravdu nastartovat by velmi pomohl při hledání chyb při načítání systému.
- **Výchozí akce při zjištěné chybě** - Pokud je program špatně nakonfigurovaný nebo v průběhu provadění změn stavu procesů nastala chyba, může systém zůstat ve stavu kdy s ním není nijak možné pracovat a je nutné ho opravit externími nástroji. V takovém případě by bylo vhodné aby Postinit alespoň spustil shell aby bylo možné se systémem pracovat.

Literatura

[1] *Linux.org*. 2021.
https://www.linux.org/.

[2] Yvan Royon a Stéphane Frénot. *A Survey of Unix Init Schemes*. 2007.
http://arxiv.org/abs/0706.2748.

[3] Matt Welsh. *Running Linux*. 4th ed vydání. O'Reilly, 2003. ISBN 978-0-596-00272-5.

[4] *Run Levels*. 2021.
https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/runlevels.html.

[5] *sysvinit - Gentoo Wiki*. 2021.
https://wiki.gentoo.org/wiki/Sysvinit.

[6] *Bash - GNU Project - Free Software Foundation*. 2021.
https://www.gnu.org/software/bash/.

[7] *su(1) - Linux manual page*. 2021.
https://man7.org/linux/man-pages/man1/su.1.html.

[8] *kill(1) - Linux manual page*. 2021.
https://man7.org/linux/man-pages/man1/kill.1.html.

[9] *systemd*. 2021.
https://www.freedesktop.org/wiki/Software/systemd/.

[10] *systemd's dependencies and installation footprint*. 2021.
https://people.debian.org/~stapelberg/docs/systemd-dependencies.html.

[11] *upstart - event-based init daemon*. 2021.
http://upstart.ubuntu.com/.

[12] *The GNU Awk User's Guide*. 2021.
https://www.gnu.org/software/gawk/manual/gawk.html.

[13] *Library sizes for C vs C++ in an embedded Linux system – BEC Systems*. 2021.
http://bec-systems.com/site/1107/library-sizes-for-c-vs-c-in-an-embedded-linux-system.

[14] *Enterprise Open Source and Linux Ubuntu*. 2021.
https://ubuntu.com/.

[15] *Welcome to Apache NetBeans*. 2021.
https://netbeans.apache.org/.

[16] *Make - GNU Project - Free Software Foundation*. 2021.
https://www.gnu.org/software/make/.

[17] Peter Goldsborough. *goldsborough/vector*. 2021.
https://github.com/goldsborough/vector.

[18] *Detect Cycle in a Directed Graph - GeeksforGeeks*. 2021.
https://www.geeksforgeeks.org/detect-cycle-in-a-graph/.

Příloha A
Zadání práce

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Štorek** Jméno: **David** Osobní číslo: **438719**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Odlehčený systém správy procesů během spouštění a ukončení chodu operačního systému Linux

Název bakalářské práce anglicky:

Light-weight process management system for bootup and shutdown of Linux OS

Pokyny pro vypracování:

Navrhněte a vytvořte podpůrnou infrastrukturu spolupracující se standardním programem SysV init. Hlavním účelem je správa systémových procesů (daemons) během všech změn úrovně běhu (Run Level) operačního systému spolu s možností manuálního zásahu (start, stop, restart, zjištění stavu procesů a další). Dodržte malou paměťovou náročnost a jen minimální závislost na dalších komponentách systému tak, aby byl produkt použitelný i v embedded systémech. / Make a design and create a supporting infrastructure cooperating with a standard SysV init. The main purpose is managing system processes (daemons) during all system run level changes together with possible manual actions like starting, stopping, restarting, status interrogation etc. Keep a low memory footprint and only minimal dependencies on other systeom compoments to let the product be usable even on embedded systems.

Seznam doporučené literatury:

[1]: Welsh, M. [et al]: Running Linux. O'Reilly Media; Fourth edition
(December 15, 2002), ISBN: 0596002726
[2]: Brodský, Skočovský: Operační systém Unix a jazyk C. SNTL 1989.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Pavel Troller, CSc., katedra telekomunikační techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2020** Termín odevzdání bakalářské práce: **21.05.2021**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Pavel Troller, CSc. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. | Datum převzetí zadání | Podpis studenta

Příloha B
Zkratky a symboly

B.1 Zkratky

PID	Process Identifier - číslo identifikující běžící proces na operační systému linux
KB	Kilobyte - jednotka velikosti dat. 1KB = 1024 byte
MB	Megabyte - větší jednotka velikost dat. 1MB = 1024KB
RAM	Random access memory - Paměť pro ukládání pracovních dat a mezivýsledků
s	sekunda, jednotka času
cmd	command, příkaz. Textová reprezentace akce kterou má systém vykonat.
IDE	Integrated Development Environment - obsáhlé prostředí pro vývoj software
MIT	The MIT License - licence pro opensource software umožňující kopírování a distribuci bez jakýchkoli omezení