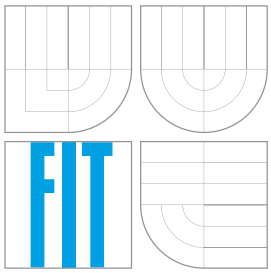
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁSTROJ PRO SESTAVENÍ VLASTNÍ DISTRIBUCE FEDORA

CUSTOM COMPILED FEDORA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN BLAŽEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2009

Zadání práce

1. Nastudujte balíčkovací systém RPM Package Manager. Nastudujte problematiku konfigurace jádra Linux.

2. Navrhněte nástroj (sadu skriptů), který umožní nainstalovat SRPM, modifikovat patřičné zdrojové soubory, výsledek sestavit a nainstalovat. Nástroj by měl také umožnit modifikovat parametry pro sestavování balíčků a dokázat optimalizovat některé vlastnosti balíčků. Cílem je vytvořit distribuci specializovanou pro daný počítač nebo hardwarové zařízení.

3. Implementujte navržené skripty.

4. Prozkoumejte možnosti modifikace modulů a parametrů jádra systému.

5. Vytvořte nástroj, který dokáže sestavit jádro na základě hardwarové konfigurace a uživatelem zadaných požadavků.

6. Zjistěte možnosti sestavení iso obrazů distribuce z modifikovaných balíčků. Vytvořte vlastní specializovanou distribuci.

Abstrakt

Tato práce se zabývá návrhem a implementací sady nástrojů pro sestavení a instalaci RPM balíčků ze zdrojových SRPM balíčků v distribuci Fedora. Nástroje umožňují modifikaci voleb kompilátoru a maker, která jsou expandována při sestavování balíčků. Součástí práce je i nástroj pro sestavení jádra Linux podle konfigurace dodané uživatelem nebo s implicitním nastavením. S použitím těchto nástrojů byla vytvořena vlastní distribuce. Výsledná distribuce obsahuje tyto nástroje pro sestavení a instalaci dalšího software.

Abstract

This thesis is focused on design and implementation of toolchain for compilation and installation RPM packages from SRPM packages in Fedora Linux distribution. Tools can be set up to modify compiler options and rpm macros. A tool for custom compilation of Linux kernel was also implemented. This tool builds the kernel with default options or according to configuration supplied by user. The toolchain was used to create customized distribution. Resulting distribution contains these tools for compilation and installation of additional software.

Klíčová slova

Linux, linuxová distribuce, Fedora, rpm, RPM balíček, SRPM, kernel, jádro, LiveCD

Keywords

Linux, Linux distribution, Fedora, rpm, RPM package, SRPM, kernel, LiveCD

Citace

Jan Blažek: Nástroj pro sestavení vlastní distribuce Fedora, diplomová práce, Brno, FIT VUT v Brně, 2009

Nástroj pro sestavení vlastní distribuce Fedora

Prohlášení

Prohlašuji, že jsem tento diplomový projekt vypracoval samostatně pod vedením pana Ing. Aleše Smrčky. Další informace mi poskytl pan Mgr. Daniel Mach z Red Hat Czech s.r.o. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jan Blažek

25. května 2009

Poděkování

Děkuji panu Ing. Aleši Smrčkovi za trpělivost a hodnotné rady během vytváření této práce. Rovněž bych chtěl poděkovat zaměstnancům firmy Red Hat Czech s.r.o, zejména panu Mgr. Danielu Machovi za konzultace.

Obsah

Kapitola 1

Úvod

Fedora je kompletní distribuce GNU/Linuxu, která je nekomerční odnoží Red Hat Linuxu. Fedora je koncipována jako univerzální distribuce a svým zaměřením se hodí spíše na osobní počítač[1].

Někdy nemusí být nasazení univerzální distribuce zrovna tím nejlepším nebo vhodným řešením. Například pro router nebo pro malý aplikační server může být požadavkem malá velikost operačního systému a pouze malý výběr nejnutnějších balíčků s minimálními závislostmi, které budou zkompilované na míru daného procesoru, a malé kompaktní jádro jen s nejnutnějšími podsystémy a ovladači.

Tato diplomová práce byla zadána společností Red Hat Czech s.r.o. Spolupráce s touto firmou a konzultace s jejími pracovníky mi umožnila nahlédnout do vývoje distribuce Fedora.

Kapitola 2 obsahuje úvod do řešené problematiky.

Kapitola 3 představuje balíčkovací systém RPM a konfiguraci jádra Linux a slouží jako teoretický úvod k dalším navazujícím kapitolám, které se na ni odkazují.

Kapitola 4 specifikuje požadavky na výsledné nástroje.

Kapitola 5 popisuje návrh nástrojů pro sestavení balíčků a sestavení jádra.

Kapitola 6 pojednává o implementaci a o sestavení výsledné distribuce.

1.1 Návaznost na semestrální projekt

V rámci semestrálního projektu byl proveden návrh nástroje pro vlastní překlad balíčků a byl implementován částečně funkční prototyp. Do diplomové práce je z něj převzata kapitola 4 – specifikace požadavků s mírnými úpravami, které jsou způsobené tím, jak jsem dané problematice začínal více rozumět a kladl jsem otázky zadavateli. Kapitola 5 – návrh byla přepracována v souladu s požadavky a s postupně objevovanými možnostmi nástrojů, na které tento projekt navazuje.

Kapitola 2

Problematika současného stavu

Operační systém GNU/Linux nemá žádný centrální model vývoje, tak jak je ho možné pozorovat u ostatních operačních systémů. Místo toho se skládá z řady komponent, které jsou vyvíjeny samostatně různými jednotlivci, komunitami nebo firmami. Základní komponentou tohoto systému je jádro Linux[1]. Kromě jádra se však do pojmu operační systém zahrnují i další jeho nezbytné části, jako jsou nejdůležitější aplikace pro konfiguraci a správu systému. Ještě z širšího pohledu je možné pod pojem operační systém zahrnout další programy, které sice nejsou důležité pro jeho samotný běh, ale uživatelé je chápou jako součást systému určenou k tomu, aby jim počítač mohl sloužit k práci nebo zábavě. Sem patří textové editory, webové prohlížeče, poštovní klienti, multimediální přehrávače a jiný software.

2.1 Distribuce

Udržovat takovýto systém v konzistentním a provozuschopném stavu při současné aktualizaci jeho jednotlivých komponent je velmi časově náročné. Proto většina uživatelů a administrátorů sáhne raději po některé z distribucí. Distribuce jsou ucelené sady softwarových balíčků spravované komunitami nebo komerčními společnostmi, které poskytují vše potřebné pro instalaci a běh systému, aktualizaci jeho součástí apod. Z hlediska architektury (potažmo i způsobu použití) je možné rozdělit distribuce do dvou základních skupin. Do první skupiny zařadím ty, které distribuují primárně zdrojové kódy programů (angl. source compiled distributions), do druhé skupiny pak ty, které poskytují již přeložené binární balíčky (programy ve spustitelné podobě pro cílovou architekturu).

2.1.1 Distribuce založené na zdrojových kódech

Distribuce, které šíří softwarové balíčky ve formě zdrojových kódů, poskytují také potřebné nástroje pro jejich kompilaci a instalaci. Tyto nástroje většinou řeší závislosti mezi jednotlivými balíčky. Zjednodušeně řečeno řeší případy, kdy pro běh programu nebo sestavení balíčku A je vyžadováno, aby v systému byl nainstalován i balíček B (např. nějaká podpůrná

[1] Někdy bývá nesprávně a zjednodušeně označován celý systém jako Linux namísto GNU/Linux

knihovna). Závislostem mezi balíčky a příslušnému teoretickému podkladu se podrobněji věnuje podkapitola 2.3.

Výhodou tohoto typu distribucí je poměrně velký prostor pro uživatelské úpravy určující podobu výsledného systému. V těchto distribucích má uživatel možnost specifikovat, jak se má daný balíček sestavovat. Může například ovlivnit, jaké konfigurační volby budou použity při kompilaci, s jakými parametry bude volán kompilátor a jaké soubory budou do výsledného balíčku zahrnuty. Typickým představitelem těchto distribucí je Gentoo[3].

2.1.2 Binární distribuce

Binární distribuce poskytují sestavené balíčky se softwarem přeloženým do strojového kódu cílové architektury. Jejich výhodou je oproti kompilovaným distribucím rychlejší instalace balíčků. Další výhodou je, že případná hlášení o chybách jsou vázána na konkrétní binární soubory, resp. na binární balíčky určité verze. Chyba tak může být reprodukována snadněji než u programů s různými modifikacemi provedenými uživatelem při sestavování. Nevýhodou je naopak menší přizpůsobitelnost výsledného systému. Systém lze na úrovni balíčků přizpůsobit pouze jejich výběrem, případně výběrem podbalíčků (angl. subpackages). Podbalíčky rozdělují seznam souborů balíčku na několik komponent, z nichž některé nemusí být nainstalovány. Jsou to například `*-doc` podbalíčky s dokumentací, `*-devel` balíčky s hlavičkovými soubory atd.[11].

2.2 Balíčky

Základní jednotkou v balíčkovacím systému jsou tzv. balíčky. Software je ve většině linuxových distribucí, ale i v jiných unixových systémech (např. FreeBSD) distribuován právě v balíčcích. Balíček je většinou komprimovaný archiv obsahující binární verzi programu a metadata, která obsahují stručný popis programu, jeho závislosti, apod.

2.2.1 Balíčkovací systémy

Balíčkovací systém je software, který si klade za cíl zjednodušit instalaci programů. Obvykle zvládá také mnoho jiných věcí spojených se správou programů v sytému. Jestliže nějaký program už nepotřebujeme, balíčkovací systém nám jej pomůže odinstalovat. Dále nám může nabídnout upgrade v případě, že vyjde nová verze programu. Při instalaci softwaru řeší některé balíčkovací systémy závislosti balíčků. Konkrétní verze programu může například záviset na jiných programech nebo knihovnách potřebných pro jeho běh. Při instalaci programu nám balíčkovací systém může nainstalovat i jeho závislosti. Pokud program chceme odinstalovat, může nám k odinstalaci nabídnout i ostatní balíčky, na kterých program závisel, ale jiné programy v systému je už nepotřebují. Různé balíčkovací systémy řeší tyto úlohy různě. Některé, jako například balíčkovací systém v distribuci Slackware, neřeší závislosti mezi balíčky vůbec a nechávají tuto práci administrátorovi.

2.3 Závislosti mezi balíčky

Jak již bylo naznačeno v podkapitole 2.1, balíčkovací systémy řeší závislosti mezi balíčky. U distribucí založených na zdrojových kódech se jedná o relace mezi jmény balíčků. Závislost může být vázána i na konkrétní verzi balíčku, používají se k tomu relační operátory `=`, `<`, `>`, `<=`, `>=`. Například zápis ve tvaru `yum >= 3.0.5` znamená, že balíček závisí na balíčku yum ve verzi 3.0.5 nebo vyšší. U binárních distribucí existují navíc kromě závislostí na balíčky i závislosti na soubory. Balíček může mít například jako závislost uveden soubor `/bin/sh` a je mu vcelku jedno, který balíček mu tento soubor poskytne, zda `bash`, `ksh` nebo jiný shell. RPM používá navíc závislosti na API. Např. závislost na `perl(XML::Parser)` značí, že balíček závisí na příslušném modulu perlu.

2.3.1 Graf závislostí

Z pohledu teorie grafů tvoří závislosti mezi balíčky graf, který má následující vlastnosti:

- Je orientovaný, protože má orientované hrany, které reprezentují vztah mezi dvěma balíčky (A závisí na B).
- Je obecný (multigraf), protože mezi dvojicí uzlů může být několik hran. Ovšem to může být někdy nežádoucí, protože to znamená, že dva balíčky na sobě vzájemně závisí. V praxi se často ukazuje, že se v tomto případě jedná o nevhodně rozdělený balíček na dva podbalíčky nebo jiné dva balíčky, které by měly být sloučeny do jednoho.
- Obvykle je nesouvislý, tj. graf tvoří jednotlivé komponenty. Většinou neexistuje žádný společný předek, na kterém by všechny balíčky závisely. I když by se za takovéhoto předka dala považovat nedílná součást systému, jako je jádro, tak se tato závislost explicitně neuvádí.
- Pokud je nesouvislý a zároveň neobsahuje kružnice, pak tvoří les.

2.3.2 Řešení závislostí

Algoritmus, který má za úkol při instalaci balíčků vyřešit jejich závislosti, funguje přibližně následujícím způsobem.

1. Vlož požadované balíčky do transakční množiny.
2. Pro všechny balíčky v transakční množině zjisti, na čem závisejí.
3. Pro závislosti zjištěné v bodu 2 najdi balíčky, které tyto závislosti poskytují a přidej je do transakční množiny.
4. Opakuj dokud je co přidávat. Pokud se dvakrát po sobě hledají ty samé závislosti, narazili jsme na chybu znamenající nesplněné závislosti.

2.3.3 Obecné podmínky

Aby řešení závislostí fungovalo korektně, musí být splněny určité obecné podmínky:

- Balíček poskytuje sám sebe (klíčové slovo `Provides` viz 3.1.3).
- Balíček daného jména smí být v systému nainstalován pouze jednou[2].

Reverzní závislosti

Při aktualizaci balíčku se nesmí zrušit závislosti jiného balíčku. Binární distribuce založené na RPM se proti tomu brání použitím položek hlavičky `Provides` a `Obsoletes`. Pomocí `Provides` dává balíček najevo, že poskytuje vlastnost daného jména. Dále balíček A může nějaký jiný balíček B deklarovat jako `Obsoletes` a dávat tím najevo, že A nahrazuje B, že B lze aktualizovat pomocí A a že naopak není možné, aby A aktualizovalo B[6].

U distribucí založených na zdrojových kódech se reverzní závislosti řeší až dodatečně po instalaci. V Gentoo se k tomu používá utilita `revdep-rebuild`[4].

[2]Speciální případ představuje jádro, které může být nainstalováno vícekrát v různých verzích. Další vyjímkou je instalace balíčků různých architektur (např. i386 a x86_64) u tzv. multilib instalací.

Kapitola 3

Balíčkovací systém RPM a konfigurace jádra Linux

Tato kapitola navazuje na studijní část zadání a přibližují se zde pojmy, které jsou používány v dalších kapitolách.

3.1 Balíčkovací systém RPM

V této podkapitole bych chtěl stručně představit balíčkovací systém RPM. Je to poměrně složitý systém, a tak se budu zabývat pouze základy a oblastmi, které přímo souvisí se zadáním této práce. Nejdříve představím balíčkovací systém RPM a potom nástroje pro sestavování balíčků.

3.1.1 Název RPM

Zkratka RPM původně znamenala Red Hat Package Manager. Jak název napovídá, stojí za RPM firma Red Hat, Inc., hlavní distributor Linuxu ve Spojených státech. Později si RPM osvojili i někteří ostatní distributoři a za zkratkou RPM dnes skrývá rekurzivní akronym RPM Package Manager[2].

3.1.2 Základní vlastnosti RPM

Binární a zdrojové balíčky

Binární balíčky obsahují již zkompilované programy. Jejich přípona bývá ve tvaru `arch.rpm`. Architekturou se rozumí architektura procesoru, pro který je balíček určen. Na procesorech od Intelu to jsou `i386`, `i586`, `i686`, a `x86_64`, na procesorech PowerPC pak `ppc` a `ppc64`.

Zvláštním případem je označení `noarch`, které je určeno pro balíčky nezávislé na architektuře. Jedná se například o interpretovaný program nebo o data nezávislá na architektuře (např. ikony).

Zdrojové balíčky, soubory s příponou `src.rpm`, obsahují zdrojové kódy a další soubory potřebné k sestavení binárních balíčků.

Čisté zdrojové kódy

RPM používá k sestavování balíčků čisté zdrojové kódy (pristine sources), tzn. zdrojové kódy v podobě, v jaké je vydává autor programu[2]. Jestliže autor RPM balíčku potřebuje zdrojové kódy upravit, vkládá k nim patch s danou úpravou. Další možností je do `spec` souboru[1] zapsat příkazy vykonávající úpravy původních zdrojových kódů.

Neinteraktivita

RPM soubor může obsahovat skripty, které jsou volány před nebo po instalaci balíčku do systému (také při upgradu balíčku). Skript může provádět činnosti jako přidávání nebo mazání uživatelských účtů a skupin, indexování knihoven `ldconfig`em, apod. Všechny tyto skripty by ale měly být neinteraktivní, aby byly operace s RPM jednoduše skriptovatelné[6].

3.1.3 Struktura spec souboru

Základem pro tvorbu nového balíčku je příprava `spec` souboru. Spec soubor definuje předpis, podle kterého se sestavují zdrojové i binární balíčky. Na začátku souboru se nachází hlavička. Její nejdůležitější položky představuje následující výpis.

Name: Jméno balíčku.

Version: Verze programu.

Release: Vydání balíčku. Začíná úrovní jedna a pokud, chce autor balíček upravit, aniž by se měnila verze programu, tak toto číslo inkrementuje o jedničku.

Summary: Jednořádkový popis balíčku.

Group: Skupina balíčku.

License: Licence programu.

URL: Domovská stránka programu.

Source0: Soubor se zdrojovým kódem programu. Píše se sice i s URL, odkud se dá přímo stáhnout, ale RPM si ho samo nestahuje a hledá ho v adresáři SOURCES. Zdrojových souborů může být i více, v tom případě se uvádějí jako další položky (Source1, Source2, atd.).

Patch0: Opravná záplata originálního zdrojového kódu. Stejně jako u Source jich může být i více (Patch1, Patch2, atd.)

[1] Podrobněji bude spec soubor popsán v sekci 3.1.3

BuildRequires: Seznam balíčků potřebných pro kompilaci balíčku. Jednotlivé položky jsou oddělené čárkami. U jednotlivých balíčků můžou být specifikovány i jejich verze (např. takto: gcc >= 4.3.2).

Requires: Totéž jako BuildRequires ale s tím rozdílem, že jsou vypsány pouze balíčky nutné pro běh programu, už ne pro jeho překlad.

Provides: Položka Provides udává, které vlastnosti (jména balíčků, API apod.) balíček poskytuje.

Ve `spec` souboru se mohou vyskytovat řádkové komentáře uvozené znakem `'#'`. Dále se zde vyskytují makra. Ty poznáme podle uvozovacího znaku `'%'`. Znakem `'%'` jsou uvozeny také základní sekce ve `spec` souboru. Těch je ale pouze malý pevně daný počet a tak je lze rozlišit od maker. Za zmínku stojí to, že makra se expandují i v komentářích[2]. A proto, chceme-li zakomentovat makro `%setup`, můžeme to udělat například takto:

```
#% setup
```

nebo

```
#%%setup
```

Sekce %description obsahuje víceřádkový popis balíčku. Tento popis se nám zobrazí, když si necháme vypsat informace o balíčku například příkazem: [2]

```
$ rpm -qi nazev_baliku
```

Sekce %prep zajišťuje rozbalení zdrojových kódů a přípravu na fázi kompilace. Se zdrojovými kódy zabalenými podle zvyklostí (tarball) si poradí makro `%setup`, které rozbalí zdrojové kódy a přejde do jejich adresáře.

```
%prep
%setup -q
```

V této sekci také mohou být aplikovány patche pomocí makra `%patch`.

Sekce %build se stará o samotný překlad. Pokud je pro konfiguraci programu použit program `autoconf`, tedy pro překlad by se volala známá trojice `configure`, `make` a `make install`, používá se volání makra `%configure`. To nastaví parametry jako `--prefix` nebo `--libdir` na hodnoty používané v distribuci. Dále toto makro nastavuje proměnné `CFLAGS` (jazyk C), `CXXFLAGS` (C++) a `FFLAGS` (Fortran) pro překladač `gcc`. Tyto proměnné pro překladač se nastavují podle makra `%optflags`.

[2] Je použita obvyklá konvence značení příkazů. Uvození znakem $ znamená, že je příkaz spouštěn pod identitou běžného uživatele. Znak # udává, že je nutné spouštět příkaz pod identitou roota.

Sekce %install slouží k nainstalování souborů do dočasného adresáře `$RPM_BUILD_ROOT`, ze kterého se potom vytvoří RPM balíček. Pro standardní `make install` můžeme využít makro `%makeinstall`.

Sekce %clean slouží k vyčištění dočasného instalačního adresáře. Obvykle se volá příkaz

```
rm -rf $RPM_BUILD_ROOT
```

Sekce %files obsahuje seznam všech souborů, které se mají zabalit do výsledného balíčku.

Sekce %changelog obsahuje komentáře změn v samotném `spec` souboru. Ve Fedoře se používá tento formát:

```
* Thu Oct 23 2008 Jan Blazek <xblaze17@stud.fit.vutbr.cz> - 0.5-1
- first version of the SPEC file
```

3.1.4 Nástroje pro sestavení balíčku

Pro sestavování balíčků je potřeba mít alespoň základní nástroje jako kompilátor gcc, make, autoconf a nástroje pro RPM. Nejjednodušeji je nainstalujeme pomocí programu `yum`:[3]

```
# yum groupinstall "Development Tools"
# yum install rpmdevtools
```

Rpmbuild

Základním nástrojem pro sestavování balíčků je `rpmbuild`. Slouží k sestavování jak binárních tak i zdrojových balíčků.

Pomocí zavolání programu `rpmdev-setuptree` můžeme v domácím adresáři vytvořit soubor `.rpmmacros` s implicitním nastavením maker a adresář `rpmbuild` s následující strukturou:

BUILD adresář, do kterého se rozbalují zdrojové kódy.

BUILDROOT adresář, kam se nainstalují soubory v sekci `%install`.

RPMS adresář, do kterého se ukládají výsledné binární RPM balíčky.

SOURCES adresář se zdrojovými kódy.

SPECS adresář se spec soubory.

SRPMS adresář, do kterého se ukládají výsledné zdrojové SRPM balíčky.

[3] Yellowdog Updater Modified - interaktivní manažer balíčků

Zdrojový balíček nainstalujeme příkazem:

```
$ rpm -i someprogram-2.0.1-2.fc10.src.rpm
```

Od verze rpm 4.6 není nutné předem volat program `rpmdev-setuptree`. Obsah zdrojového balíčku se standardně nainstaluje do `$HOME/rpmbuild`.

Konfiguraci `rpm` a `rpmbuildu` můžeme měnit v souboru rpmrc. Tento soubor můžeme programu `rpm` vnutit pomocí parametru `--rcfile` nebo se použijí rpmrc soubory v systému. Většinou seznam těchto souborů vypadá takto:

```
/usr/lib/rpm/rpmrc:/usr/lib/rpm/redhat/rpmrc:/etc/rpmrc:~/.rpmrc
```

Jednotlivé soubory mají různou prioritu. V tomto případě je priorita prvního uvedeného souboru `/usr/lib/rpm/rpmrc` nejmenší a u `~/.rpmrc` (v seznamu uvedeném na konci) největší. V souboru `/usr/lib/rpm/rpmrc` se nachází globální nastavení. V `/etc/rpmrc` můžeme definovat nastavení specifická pro systém a v `~/.rpmrc` nastavení pro samotného uživatele.

Pro nás je v souboru `rpmrc` nejzajímavější proměnná *optflags*. Ta definuje volby kompilátoru. V tomto případě na procesoru třídy i686 necháme gcc, aby si samo upravilo parametry kompilace podle konkrétního procesoru, na kterém se provádí kompilace:

```
optflags: i686 -O2 -march=native
```

Mock

Mock je nástroj, který pro kompilaci zdrojových SRPM balíčků vytváří chroot prostředí[4]. Toto prostředí je naplněno všemi soubory potřebnými pro sestavení balíčku. Nakopírují se do něj základní programy a knihovny a nainstalují se také všechny balíčky, které jsou uvedeny v hlavičce spec souboru v položce BuildRequires[12].

Důvodem existence mocku je právě izolované chroot prostředí. Do systému se tak nemusí instalovat balíčky obsahující nástroje a knihovny potřebné pro sestavování RPM balíčků. Oddělení chrootu od zbytku systému by také mělo umožnit opakovat sestavení balíčku s víceméně stejnými výsledky na jiném stroji nebo po výraznějších zásazích do systému.

3.2 Konfigurace jádra Linux

Tato kapitola popisuje možný způsob sestavení vlastního linuxového jádra a vychází přitom z návodu pro distribuci Fedora[10]. Důvodů proč kompilovat vlastní jádro může být několik:

- Nutnost použít funkce, které nejsou zakompilovány v distribučním jádře.
- Potřebujeme nějakou funkci, která se ještě nedostala do hlavního stromu jádra a je k dispozici pouze jako patch.

[4] Příkaz chroot slouží ke změně kořenového adresáře (change root) pro proces a jeho potomky. Proces je tak na úrovni souborového systému izolován.

- Chceme si to vyzkoušet a naučit se něco nového o systému GNU/Linux.

3.2.1 Příprava

Tato sekce popisuje přípravu zdrojových kódů jádra od stažení zdrojového balíčku přes jeho instalaci a přípravu stromu zdrojových kódů jádra.

Prerekvizity

Jako prerekvizity potřebujeme mít nainstalovány balíčky `yum-utils` a `rpmdevtools`. Nainstalujeme je příkazem:

```
# yum install rpmdevtools yum-utils
```

Příprava zdrojových kódů

1. Zdrojový balíček pro sestavení jádra můžeme stáhnout příkazem:

   ```
   $ yumdownloader --source kernel
   ```

2. Nástroj dokáže zjistit, které balíčky jsou potřeba k sestavení daného zdrojového balíčku a umí je i nainstalovat.

   ```
   # yum-builddep kernel-<verze>.src.rpm
   ```

3. Nainstalujeme zdrojový balíček. Ve Fedoře verze 10 se standardně nainstaluje do `$HOME/rpmbuild`.

   ```
   $ rpm -Uvh kernel-<verze>.src.rpm
   ```

4. Necháme rpmbuild provést přípravnou fázi, tj. rozbalení zdrojových kódů, aplikaci patchů apod.

   ```
   $ cd ~/rpmbuild/SPECS
   $ rpmbuild -bp --target=‘uname -m‘ kernel.spec
   ```

3.2.2 Konfigurace

1. Vstoupíme do adresáře se zdrojovými kódy jádra:

   ```
   cd ~/rpmbuild/BUILD/kernel-<verze>/linux-<verze>.<arch>
   ```

2. Může být jednodušší vycházet z nastavení distribučního jádra a pouze měnit potřebné položky než vytvářet celý konfigurační soubor od základů.

   ```
   $ cp configs-<arch> .config
   ```

Jestliže nechceme vycházet z distribučního konfiguračního souboru, můžeme použít příkaz `make defconfig`, který nastaví volby v konfiguraci na implicitní hodnoty pro danou architekturu, podle toho, jak je definovaly tvůrci jádra.

Příkaz `make oldconfig` projde konfigurační soubor `.config`, který může například obsahovat starší konfiguraci, a nastaví nové konfigurační volby podle něj.

3. Následujícím příkazem spustíme konfigurační nástroj, pomocí kterého můžeme nastavit požadované volby. Tento nástroj funguje v textovém režimu a k vykreslování používá knihovnu ncurses (viz obr. 3.1).

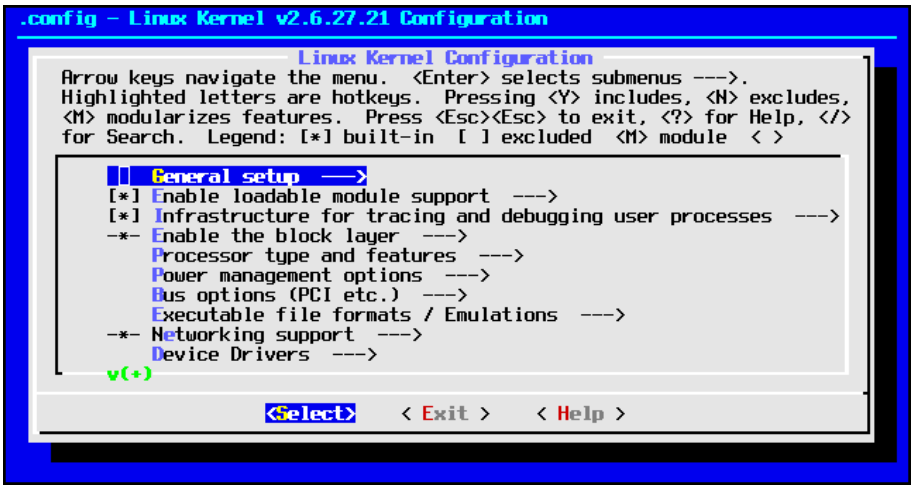
```
$ make menuconfig
```

   - Alternativně můžeme použít grafické konfigurační rozhraní, které využívá knihovnu Qt:

     ```
     $ make xconfig
     ```

   - Jestliže dáváme přednost spíše GTK, můžeme vyzkoušet:

     ```
     $ make gconfig
     ```



Obrázek 3.1: Konfigurační nástroj menuconfig

4. Výsledný konfigurační soubor zkopírujeme do adresáře SOURCES:

```
$ cp .config ~/rpmbuild/SOURCES/config-<arch>
```

3.2.3 Sestavení balíčku a instalace

1. Vstoupíme do adresáře se spec soubory

   ```
   $ cd ~/rpmbuild/SPECS
   ```

2. Abychom odlišili námi sestavené jádro od distribučního jádra, je potřeba v souboru `kernel.spec` přepsat řádek

   ```
   #% define buildid .local
   ```

   na

   ```
   %define buildid .<zvoleny_nazev>
   ```

3. Můžeme přikročit k sestavení RPM balíčků jádra pomocí nástroje rpmbuild.

   ```
   $ rpmbuild -bb --target=‘uname -m‘ kernel.spec
   ```

   Sestavením balíčku vznikne několik variant jádra (běžné jádro, jádro se zapnutým PAE[5], atd.). Sestavení rpm balíčku můžeme výrazně urychlit, pokud budeme sestavovat pouze jednu variantu. Provedeme to přidáním volby např. `--with baseonly` nebo `--with paeonly` pro program `rpmbuild`. Tyto argumenty se expandují do podoby maker (např. `%_with_baseonly 1`), která jsou kontrolována v podmíněných sekcích ve spec souboru.

   ```
   $ rpmbuild -bb --with baseonly --target=‘uname -m‘ kernel.spec
   ```

4. Výsledný balíček jádra nainstalujeme.

   ```
   # rpm -ivh ~/rpmbuild/RPMS/<arch>/kernel-<verze>.<arch>.rpm
   ```

[5] Physical address extension = režim rozšíření fyzické adresy nad 4GB na 32-bitové architektuře Intel IA-32

Kapitola 4

Specifikace požadavků

V této kapitole se pokusím rozvést jednotlivé požadavky na aplikaci, popíšu jednotlivé akce, které má aplikace provádět. Vymezením konkrétních požadavků se přiblížím návrhu.

4.1 Nástroj pro sestavení balíčků

Distribuce se skládá především z balíčků. V případě distribuce Fedora se jedná o balíčky typu RPM. Nástroj by měl umožnit sestavit balíčky podle uživatelem specifikovaných parametrů, které by předepisovaly modifikace. Modifikacemi balíčků jsou myšleny parametry jejich sestavování. Správně by vlastnosti, které nejsou pro balíček klíčové, jako jsou například parametry konfiguračního skriptu `configure` (případně jeho ekvivalentu, pokud se konfigurace neprovádí programem autoconf) nebo závislosti, měly jít vypnout pomocí maker při překladu. Používají se k tomu makra `%with` a `%without`, pomocí nichž jsou ve spec souboru zapsány podmíněné sekce, které vypínají nebo naopak zapínají různé volby[6].

Další změnou v sestavování balíčku je nastavení voleb pro kompilátor. V distribuci Fedora je dnes na architektuře x86 většina balíčků zkompilována na instrukční sadu i386[1], pouze některé jako kernel, glibc, aj. jsou přeloženy pro instrukční sadu i686. Takto přeložené balíčky jsou sice binárně kompatibilní s procesory i386 resp. Pentium II (i686), ale na druhou stranu nemohou využít bohatší instrukční sadu novějších procesorů. Překladem programů moderním kompilátorem, jakým je gcc, můžeme po zapnutí správných voleb docílit toho, že se bude generovat rychlejší kód obsahující instrukce z rozšířených instrukčních souborů jako MMX, SSE, SSE2 apod. Dále je možné zapnout různé stupně a způsoby optimalizace na úrovni kompilátoru.

Z hlediska zdrojů pro balíčky se nabízí dva případy. Za prvé se jedná o balíčky, které poskytuje distribuce ve svých oficiálních repozitářích, případně které poskytují repozitáře třetích stran. U těchto balíčků může uživateli stačit překompilovat je s jinými parametry kompilátoru a s pozměněným nastavením maker, aby docílil požadovaného efektu. Dalším zdrojem může být sada vlastních zdrojových balíčků. Může se jednat například o balíčky,

[1] Fedora verze 11 už používá pro většinu balíčků instrukční sadu i586

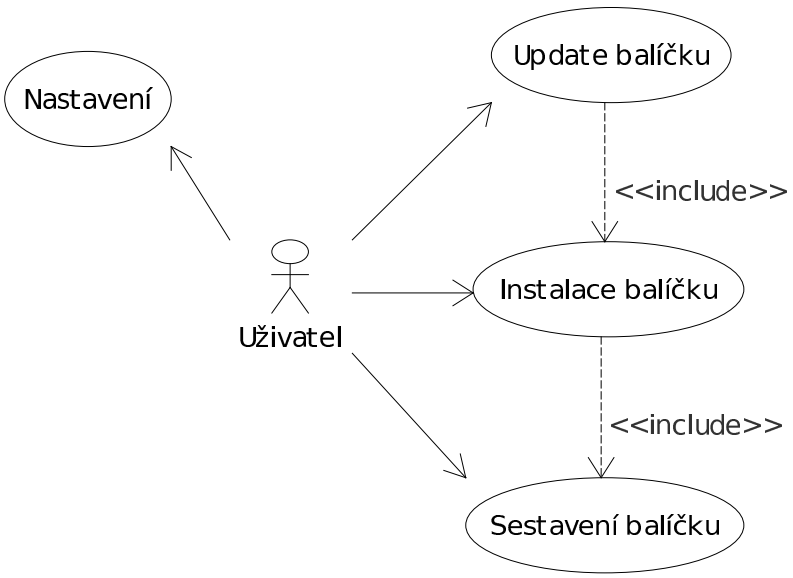
které nejsou z licenčních důvodů distribuovány přes oficiální repozitáře distribuce.

Nástroj by měl umožnit stažení zdrojového balíčku (SRPM) a z něj sestavit výsledný binární balíček. Zdrojové balíčky je možné stahovat přímo z repozitářů distribuce Fedora. Otázkou potom je, jak se staženými balíčky zacházet. Nabízí se možnost balíčky uchovávat v nějakém k tomu určeném adresáři, aby se nemusely znovu stahovat, pokud by je uživatel chtěl znovu použít pro překlad balíčku například s jiným nastavením. Bylo by vhodné mít možnost omezit velikost tohoto odkládacího adresáře a mazat starší soubory, pokud by celková velikost souborů překročila danou hranici. Tuto hranici by bylo možné nastavit v konfiguračním souboru. Další možností je při stahování zdrojového balíčku kontrolovat, zda se v odkládacím adresáři nenachází starší verze téhož a případně ji smazat.

Další otázkou je, jak aktualizovat tyto pozměněné balíčky. Bylo by vhodné, aby při aktualizaci systému nebyly modifikované balíčky přepsány výchozími balíčky z distribuce. Jednoduchým řešením by bylo u těchto balíčků zakázat jejich aktualizaci úplně a překlad a instalaci novější verze nechat na ruční práci administrátora systému. Vhodnější by však bylo mu tuto činnost zjednodušit. Jednou z možností by mohlo být vytvoření zásuvného modulu (angl. plugin) pro nástroj `yum`. Zásuvný modul by v případě aktualizace modifikovaného balíčku sám mohl zavolat nástroj pro překlad balíčku a podle nastavených parametrů by vytvořil nový binární balíček.

Výsledné binární balíčky je vhodné uchovávat pro další použití. Nabízí se možnost vytvořit pro ně speciální lokální yum repositář a do něj tyto balíčky umísťovat. Vhodným nastavením yumu by se dalo docílit toho, aby měl tento repozitář vetší prioritu a aby tak yum instaloval balíčky přednostně z něj.

Základní případy užití nástroje jsou zobrazeny na diagramu 4.1.



Obrázek 4.1: Diagram případů užití

4.1.1 Funkční požadavky

Nástroj by měl splňovat následující požadavky shrnuté do několika bodů:

Stažení zdrojového RPM

- stáhnout z repozitáře zdrojové RPM (SRPM)
- uložit SRPM do vhodného adresáře
- nějakým způsobem spravovat stažené SRPM – podle nastavení např. mazat staré balíčky

Překlad

- sestavení binárního balíčku
  - se zadaným nastavením parametrů kompilátoru
  - s dalším nastavením voleb překladu pomocí maker

Instalace

- instalace vytvořeného balíčku

Aktualizace

- umožnit aktualizaci systému a balíčků sestavených tímto nástrojem

Nastavení

- nastavení specifikováno v konfiguračním souboru
- upřesnit, které balíčky se mají sestavovat pomocí tohoto nástroje
- nastavení parametrů překladu ve dvou úrovních:
  - implicitní nastavení pro všechny kompilované balíčky (typicky výchozí parametry kompilátoru)
  - specifické nastavení s vyšší prioritou pro konkrétní balíčky (další nastavení překladu pomocí maker)
- nastavení způsobu mazání cache SRPM balíčků
- případné další možnosti nastavení přes parametry příkazové řádky

4.2 Nástroj pro sestavení jádra

Jedním z bodů zadání bylo vytvořit nástroj pro usnadnění překladu linuxového jádra. Někdy totiž konfigurace jádra, tak jak je nastavena v distribuci, nemusí uživateli vyhovovat. Uživatel si může přát například jádro na server, kde z důvodu bezpečnosti je úplně vypnut usb subsystém, aby se nedalo připojit žádné usb zařízení apod.

Distribuční jádra mají typicky zakompilovánu spoustu ovladačů ve formě modulů, aby bylo jádro co nejvíce univerzální ve smyslu použití na širokém spektru HW. Při konfiguraci vlastního jádra se však vyplatí nepotřebné ovladače pro HW, který nemáme, úplně vypnout. Výrazně se tím zrychlí doba překladu jádra.

Kapitola 5

Návrh

Tato kapitola se zabývá návrhem nástrojů. Jsou zde popsány jednotlivé součásti nástrojů, které slouží k dosažení požadované funkčnosti. Nejdříve je popsán diagram tříd pro lepší celkovou představu o vzájemných vztazích mezi jednotlivými komponentami. Potom je věnován prostor k představení jednotlivých komponent.

5.1 Architektura aplikace

V této části se pokusím popsat architekturu navrhované aplikace. Navrhovaná aplikace se bude skládat z několika vzájemně se doplňujících nástrojů. Tyto nástroje mají za úkol skloubit již existující nástroje v distribuci Fedora a usnadnit uživateli práci při vlastním sestavování balíčků a při překladu jádra.

5.1.1 Diagram tříd

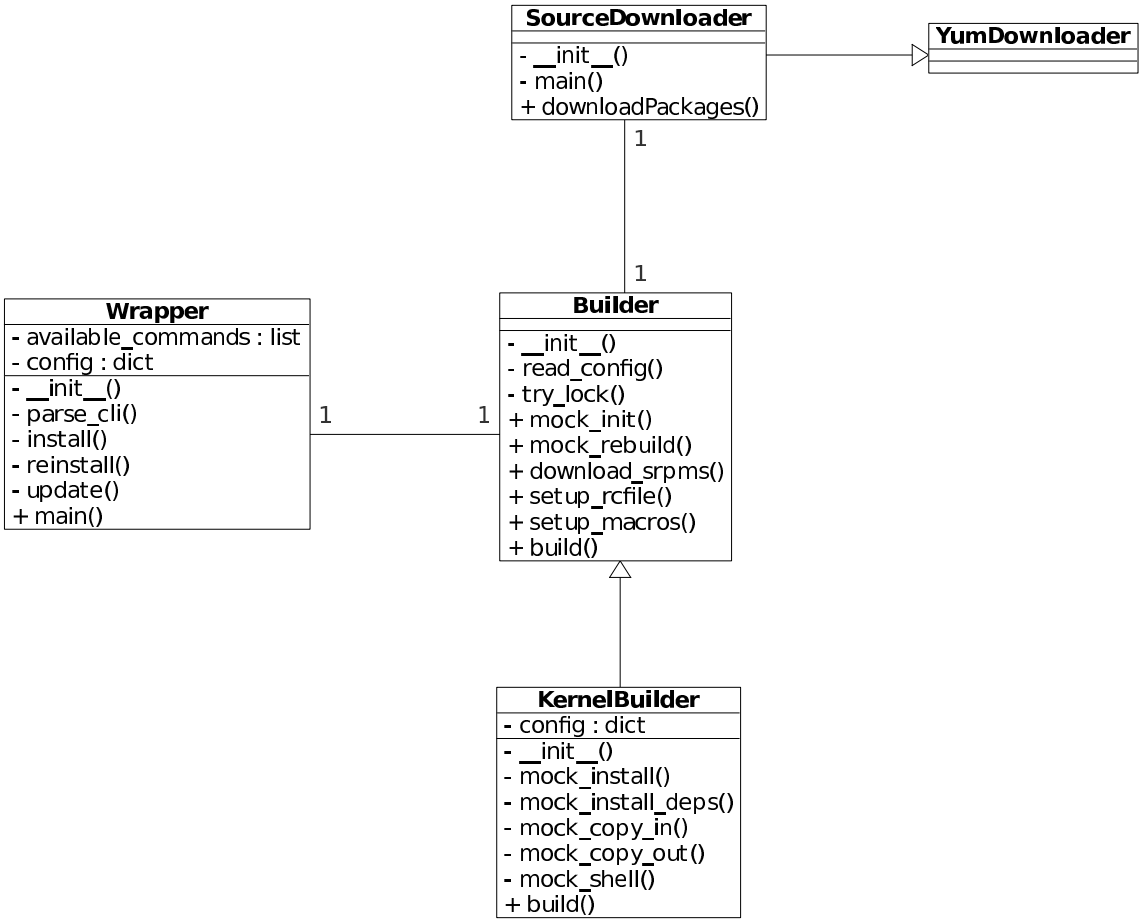
Následující diagram 5.1 znázorňuje jednotlivé třídy, ze kterých se budou výsledné nástroje skládat. Poté následuje výpis těchto tříd s popisem u každé třídy, který vysvětluje její účel.

Builder Hlavní třída, která se stará o sestavení balíčku podle nakonfigurovaných parametrů. Volá metody ostatních tříd a další pomocné funkce.

KernelBuilder Třída pro překlad jádra má mnoho společného s třídou **Builder** a proto je od ní odvozená. Může tak používat některé metody této třídy, případně některé metody může předefinovat vlastními metodami pro dosažení požadovaného chování.

Wrapper Třída, která slouží k obalení aplikace `yum` pro účely instalace, reinstalace nebo update balíčku.

SourceDownloader Třída, která slouží pro stahování zdrojových balíčků z nastavených repozitářů. Je odvozená od třídy **YumDownloader**, kterou poskytuje externí nástroj `yumdownloader(1)`.



Obrázek 5.1: Diagram tříd

5.1.2 Další funkce

Kromě metod základních tříd nástroje budou potřeba ještě další funkce spíše pomocného charakteru. Sem patří například podpora pro spouštění programů jako podprocesů hlavního programu a čtení jejich výstupu. Dále různé funkce pro práci s repozitáři, s rpm soubory, funkce pro čtení informací z RPM balíčků, apod.

5.1.3 Konfigurace

Vzhledem k provázanosti jednotlivých nástrojů, které budou využívat společné komponenty je vhodné, aby tyto nástroje měli společnou konfiguraci. V operačním systému GNU/Linux se pro konfiguraci nástrojů používají nejčastěji textové soubory umístěné v adresáři `/etc`. Jelikož implementované nástroje budou navazovat na jiné nástroje, je třeba zajistit i konfiguraci těchto nástrojů přes jejich konfigurační soubory, nebo pokud to umožňují, může být výhodnější konfiguraci vynutit dočasně předáním přes parametry příkazové řádky.

5.2 Návrh nástroje pro sestavení balíčků

Tato sekce představuje návrh nástroje pro sestavení balíčků. Nejdříve je uvedena základní idea, jak by mohl nástroj vnitřně fungovat a potom je věnován prostor popisu jednotlivých metod a funkcí.

Základní myšlenkou pro návrh tohoto nástroje je využít pro sestavování balíčků program `mock(1)`, představený v kapitole 3 v sekci 3.1.4. Balíčky se tak budou sestavovat v prostředí izolovaném od ostatního systému a nebude potřeba do systému instalovat programy, které uživatel běžně nechce (a které jsou využitelné jenom při sestavování balíčků, jako jsou různé překladače, `*-devel` podbalíčky s hlavičkovými soubory, apod.)

5.2.1 Inicializace mocku

V základním nastavení je mock při každém spuštění znovu inicializován a uveden do základního čistého stavu, takže pro překlad jednotlivých balíčků jsou v chroot prostředí nainstalovány pouze balíčky potřebné pro jejich sestavení[9]. Mock často využívají správci balíčků (angl. package maintainer), aby zkontrolovali, zda ve spec souboru uvedli správně všechny BuildRequires položky (viz 3.1.3). V tom případě je vhodné, aby byl mock pro překlad každého balíčku znovu inicializován a aby na počátku bylo v chrootu nainstalováno jen základní prostředí. Samotná inicializace mocku je však časově náročná. Pro urychlení práce nástroje pro sestavování balíčků znatelně pomůže, když se bude mock inicializovat pouze jednou na začátku dávky. Při sestavování více balíčků najednou se tak zdržení při inicializaci rozmělní mezi celkovou dobu potřebnou pro jejich sestavení.

5.2.2 Lokální repozitář

Jednotlivé sestavené balíčky mohou být během jedné dávky průběžně ukládány do lokálního repozitáře, ze kterého si mock bude brát balíčky potřebné pro sestavení dalších balíčků. Aby byly balíčky instalovány přednostně z tohoto lokálního repozitáře a ne z jiných nakonfigurovaných repozitářů, je nutné u něj nastavit vyšší prioritu v konfiguračním souboru. Aby byly nově sestavené balíčky rozpoznatelné pro yum, nestačí je pouze nakopírovat do repozitáře, ale je také potřeba aktualizovat metadata repozitáře. K tomu slouží utilita `createrepo(8)`. Yum si však metadata stažená z repozitářů ukládá do vlastní cache, aby je nemusel příliš často stahovat ze vzdálených serverů. Během sestavování několika balíčků se však metadata u lokálního repozitáře budou měnit poměrně často, a proto je potřeba nakonfigurovat yum tak, aby si žádnou cache pro tato metadata nevytvářel.

5.2.3 Závislosti mezi balíčky

Pro řešení závislostí při instalaci binárních balíčků je v distribuci Fedora k dispozici yum. Yum slouží pouze k instalaci binárních balíčků. Depsolver[1] v yumu však řeší pouze závislosti mezi `Provides` a `Requires` binárních balíčků a stačí mu vytvořit pouze množinu balíčků se vzájemně splněnými závislostmi. Když se výsledná množina nainstaluje, neměla by nastat situace, kdy nějaký program potřebuje ke svému běhu například nějakou knihovnu a daná knihovna se v systému nenachází.

Naproti tomu při sestavování balíčků potřebujeme znát závislosti mezi `BuildRequires` a `Provides`, aby balíček který je právě sestavován měl k dispozici všechny balíčky, které ke svému sestavení potřebuje. Všechny `Provides` však neznáme do té doby, než balíčky sestavíme. Nestačí nám ani množina balíčků se splněnými závislostmi, ale potřebujeme znát pořadí jejich sestavení.

Dalším problémem, se kterým je třeba se vypořádat, je to, že z jednoho SRPM může být sestaveno více RPM. Mohou to být podbalíčky ale i balíčky s úplně jiným jménem, než má zdrojové SRPM.[2] Informaci o tom, které balíčky vzniknou sestavením ze zdrojového SRPM nelze vyčíst z jeho hlavičky ale pouze ze spec souboru. Musíme tedy mít k dispozici celé SRPM a ne jenom jeho hlavičku, která se ukládá do metadat yum repozitáře.

Návrh řešení závislostí

Položky `BuildRequires` je možné vyextrahovat z hlaviček SRPM. Za `Provides` můžeme s jistými omezeními považovat jména podbalíčků, které vzniknou sestavením daného SRPM a v hlavičce SRPM explicitně uvedené `Provides`. Jména zdrojových balíčků s jejich příslušnými `Provides` a `BuildRequires` se vloží na vstup algoritmu, který má za úkol zjistit pořadí sestavení balíčků.

Tento přístup má však omezení v tom smyslu, že vlastní SRPM balíčky (tedy ty, které nejsou převzaty z distribuce) by neměly mít `BuildRequires` závislosti na soubory nebo jiné závislosti, které se generují až při sestavování RPM balíčku. U SRPM balíčků získaných z distribuce špatně zjištěné pořadí sestavování nevadí, protože v případě potřeby se jako závislost potřebná k sestavení balíčku může použít distribuční RPM balíček. Zrovna tak se pro nesplněné závislosti z výstupu následujícího algoritmu mohou použít RPM balíčky z distribuce.

Algoritmus pro určení pořadí sestavování balíčků

Vstup: seznam balíčků

Výstup: uspořádaný seznam nalezených závislostí *found*, seznam nesplněných závislostí *not_found*

[1] dependency resolver neboli modul pro řešení závislostí

[2] Např. SRPM glibc je zdrojem pro balíček nscd.

Jednotlivé kroky:

1. Vlož do zásobníku *s* jména balíčků
2. Pokud není zásobník *s* prázdný, vlož obsah vrcholu zásobníku *s* do proměnné *top*, jinak jdi na bod 6.
3. Hledej v *provides*, které balíčky poskytují vlastnost *top* a vlož je na začátek seznamu nalezených závislostí *found*, pokud v něm ještě nejsou.
4. Pokud nebyl nalezen žádný balíček, který by poskytoval vlastnost *top*, vlož *top* do seznamu nesplnitelných závislostí *not_found* a pokračuj bodem 2.
5. Zjisti, na čem závisí *top* a vlož všechny tyto závislosti, které nejsou v seznamu nesplnitelných závislostí *not_found*, na zásobník *s*. Pokračuj bodem 2.
6. Skonči. Výstup je v seznamech *found* a *not_found*.

5.2.4 Konfigurační soubory

Nastavení nástrojů se provádí na několika úrovních. Nejdříve je to hlavní konfigurační soubor společný pro více nástrojů, dále jsou to nastavení pro programy, které jsou jednotlivými nástroji volány. Mělo by jít také specifikovat volby překladu jednotlivých balíčků.

Hlavní konfigurační soubor

Hlavní konfigurační soubor obsahuje společná nastavení sady nástrojů pro překlad balíčků. Jako vhodné řešení se jeví použít syntaxi, se kterou si poradí modul ConfigParser z jazyka Python. Struktura takovýchto konfiguračních souborů je podobná struktuře INI souborů známých z Microsoft Windows. Konfigurace v tomto formátu sestává ze sekcí uvozených řádkem `[sekce]`, které obsahují položky `název = hodnota`. Řádky začínající znakem `#` (mřížka) nebo `;` (středník) jsou považovány za komentáře[7]. Tato hlavní konfigurace nástroje bude umístěna v souboru `/etc/pkgbuild/config`.

Konfigurace maker

Pro jednotlivé balíčky by mělo být možné nastavit makra. Typicky se bude jednat o nastavení voleb pomocí maker `%with` a `%without`, viz kap. 4. Jako vhodné řešení se jeví mít pro každý balíček daného jména speciální soubor s makry, který později využije `rpmbuild` v mocku tak, že se z něho vytvoří soubor `rpmmacros`. Je asi zbytečné, aby každý balíček měl svůj vlastní konfigurační soubor i v případě, že žádné speciální nastavení maker není pro něj vyžadováno. V případě, že nástroj nenalezne soubor s makry pro daný balíček, použije se soubor s názvem `_default`, který bude obsahovat implicitní hodnoty. Soubory s konfigurací maker budou umístěny v adresáři `/etc/pkgbuild/macros`.

Konfigurace pro rpmbuild

Pro konfigurační soubory `rpmbuildu`, ve kterých se nastavuje například položka `optflags` (viz. 3.1.4), se dá použít podobné schéma jako u konfigurace maker. Ve speciálním adresáři se budou nacházet soubory pojmenované podle balíčků, které se použijí jako `rpmrc` soubory pro `rpmbuild`. Pro případ implicitního nastavení se použije soubor `_default`. Tyto konfigurační soubory se budou nacházet v adresáři `/etc/pkgbuild/rpmrc`.

Konfigurace mocku

Jelikož navrhovaný nástroj využívá pro překlad mock, je potřeba správně nastavit konfigurační soubor mocku. Důležitými volbami, které je potřeba nastavit jsou:

- Název chrootu, který udává jméno adresáře, který bude mockem využit jako chroot. Typicky se nachází v adresáři `/var/lib/mock`.
- Značka distribuce neboli dist tag (např. `fc10`), která se připojuje k názvu výsledných rpm souborů.
- Položka `resultdir`, která udává cestu k adresáři, do kterého se budou ukládat výsledné RPM balíčky.
- Nastavení zásuvného modulu `bind_mount`, který umožňuje připojit podstrom adresářové struktury v systému do adresáře v chrootu. Toto nalezne využití pro zpřístupnění lokálního yum repozitáře pro mock.
- Nastavení cesty k lokálnímu repozitáři, zvýšení jeho priority oproti ostatním repozitářům, tak aby byly balíčky instalovány přednostně z něho a zákaz vytváření cache metadat tohoto repozitáře.

5.2.5 Instalace a aktualizace balíčků

Sestavené balíčky se nacházejí v lokálním repozitáři (viz. 5.2.2), a tak je možné je nainstalovat pomocí programu `yum`. Pro snadnější instalaci je však možné vytvořit wrapper[3], který balíčky nejdříve sestaví a potom zavolá yum, aby je nainstaloval. Wrapper může mít stejné parametry volání jako yum s tím rozdílem, že u některých příkazů provede navíc akci související se sestavováním balíčků. Příkazy, které může wrapper zpracovávat jsou `install`, `reinstall` a `update`. Pokud je wrapper zavolán s jinými argumenty, můžou být tyto argumenty v nezměněné podobě předány programu yum.

install Aby byla zachována sémantika příkazu `install` stejná, jaká je u programu yum, dojde nejprve ke kontrole, zda balíčky daných jmen nejsou v systému už nainstalovány.

[3]Wrapper program obalující volání jiného programu, viz slovníček. Dále bude v textu používáno slovo wrapper.

Pokud ano, wrapper se ukončí s informativní hláškou, stejně jako je to u programu yum. Jinak jsou požadované balíčky sestaveny a je zavolán yum pro jejich instalaci.

reinstall Příkaz `reinstall` funguje podobně jako `install` s s tím rozdílem, že podmínka je zde obrácená. Pokud jsou balíčky daného jména již nainstalovány, dojde k jejich opětovnému sestavení a instalaci.

update Příkaz `update` by měl aktualizovat balíčky. Syntaxe je podobná jako u programu yum. Pokud za příkazem `update` následují argumenty v podobě jmen balíčků, jsou sestaveny a aktualizovány pouze vybrané balíčky. Při volání bez dalších argumentů jsou aktualizovány všechny balíčky, které mají k dispozici novější zdrojové SRPM balíčky v lokálním repozitáři.

5.2.6 Popis jednotlivých tříd a jejich metod

Třída Build

Třída Build je hlavní třídou nástroje pro sestavení balíčků. Je využitelná pro samostatný nástroj, který umí balíčky pouze sestavit, nebo nalezne využití například pro wrapper, který dokáže sestavené balíčky i nainstalovat.

__init__() Konstruktor, ve kterém se provede inicializace třídy a proměnné udávající nastavení aplikace se naplní hodnotami z konfiguračního souboru pomocí *read_config()*.

read_config() Tato metoda přečte nastavení z konfiguračního souboru, zároveň zkontroluje jeho syntaxi.

try_lock() Metoda sloužící jako globální zámek aplikace. Vzhledem k tomu, že v daném chrootu může běžet pouze jedna instance mocku, je vhodné zajistit, aby i nástroj využívající mock mohl běžet pouze v jedné instanci. Tato metoda je volána při inicializaci aplikace z konstruktoru. Jestliže se zjistí, že zámek je už jiným procesem zamčen, aplikace se ukončí s informativní hláškou vypsanou do konzole.

mock_init() Metoda, která provede inicializaci mocku.

mock_rebuild() Tato metoda zavolá mock pro sestavení jednoho SRPM balíčku.

setup_rcfile() Provede nastavení rpmrc souboru pro SRPM daného jména.

setup_macros() Nastaví RPM makra pro SRPM daného jména.

build() Hlavní metoda třídy pro sestavování balíčků. Vstupem je seznam jmen balíčků, které mají být sestaveny. Provede se zjištění pořadí, ve kterém je nejlepší balíčky sestavovat a balíčky jsou postupně sestaveny.

Třída SourceDownloader

Program `yumdownloader` z balíčku `yum-utils` slouží ke stahování balíčků ze vzdálených repozitářů. Program `yumdownloader` volaný s parametrem `--source` umí stáhnout z nastavených repozitářů zdrojové SRPM balíčky k binárním RPM balíčkům, jejichž jména jsou mu předána jako argumenty. Při použití tohoto programu ve skriptu mu potřebujeme předat parametry udávající, co má stáhnout a potom potřebujeme zjistit, co skutečně stáhl. Naneštěstí však `yumdownloader` neposkytuje dostatečně přehledný výstup, který by byl dobře zpracovatelný v našem programu. Možným řešením, jak se s tímto vypořádat, je vytvořit si vlastní třídu, která bude odvozená od třídy YumDownloader a předefinovat si některé metody vlastními metodami s požadovaným chováním. Potřebujeme předefinovat tyto metody:

__init__() V konstruktoru přidáme pouze nastavení vlastních parametrů, které se v aplikaci `yumdownloader` provádí nastavením parametrů z příkazové řádky.

main() V metodě main potřebujeme zrušit čtení argumentů z příkazové řádky.

downloadPackages() Tato metoda může mít stejné chování jako původní metoda **downloadPackages**, s tím rozdílem, že vrací údaje o stažených balíčcích. Vstupem je seznam jmen RPM balíčků a výstupem je seznam stažených SRPM balíčků.

Třída Wrapper

Třída Wrapper používá pro sestavení balíčků třídu `Builder` a slouží jako rozhraní k volání programu `yum`.

__init__() Konstruktor nastaví některé atributy třídy podle údajů z konfiguračního souboru.

parse_cli() Tato metoda má za úkol předzpracovat argumenty příkazové řádky. Argumenty jsou rozděleny do tří částí podobně, jako tomu je u programu `yum(8)`. V první části jsou nepovinné parametry nastavující různé volby, které mohou být nastaveny i v konfiguračním souboru `yum.conf`. Další částí je příkaz update, install, atd. A ve třetí části mohou být názvy balíčků.

main() Hlavní funkce, která slouží pro zpracování argumentů příkazové řádky. Volá příslušné metody wrapperu. Pokud byl wrapper spuštěn s neznámým příkazem, jsou všechny argumenty předány v nezměněné podobě yumu.

install() Tato metoda má za úkol zjistit, zda balíčky k instalaci nejsou v systému už nainstalovány. Pokud tomu tak není, dojde ke stažení zdrojových SRPM a k jejich sestavení, o které se postará instance třídy **Builder**.

reinstall() Metoda reinstall slouží k opětovnému překladu a instalaci balíčků, které už jsou v systému nainstalovány.

update() Metoda update může být, podobně jako příkaz `yum update`, volána s argumenty v podobě balíčků, které se mají aktualizovat. Při volání bez argumentů se provede aktualizace všech nainstalovaných balíčků. Funkce funguje tak, že pro všechny balíčky v lokálním repozitáři (případně pouze pro dané balíčky při volání s argumenty) zkusí stáhnout nové SRPM balíčky. Pokud se v lokálním SRPM repozitáři najdou SRPM balíčky novější, než jsou nainstalované, dojde k jejich sestavení. Nakonec se zavolá příkaz `yum update`.

5.3 Návrh nástroje pro sestavení jádra

Jedním z bodů zadání bylo vytvořit nástroj, který by měl uživateli pomoci při sestavování linuxového jádra. Požadavky na tento nástroj vycházejí ze způsobu sestavení vlastního jádra, který je popsán v podkapitole 3.2. Je to způsob, který jako zdroj nevyužívá čisté zdrojové kódy z `http://www.kernel.org`, jak je to popisováno v některých návodech pro sestavení jádra, ale zdrojové SRPM poskytované distribucí. Tento postup má několik výhod:

- Při konfiguraci můžeme vycházet z nastavení distribučního jádra.
- Výsledkem je RPM balíček. To přináší řadu výhod s tím spojených, o instalaci (včetně nastavení zavaděče systému) nebo případné odinstalování se postará program pro správu balíčků.

Tento nástroj má mnoho společného s předešlým nástrojem, protože se jedná o sestavení RPM balíčku ze zdrojového SRPM. Má však i některá specifika, a proto se zdá být vhodné, aby třída této aplikace byla potomkem třídy **Builder** pro sestavení balíčků. V nové třídě může být předefinováno chování původních metod a navíc mohou být implementovány další speciální metody.

5.3.1 Architektura nástroje

Nástroj pro překlad jádra by měl automatizovat jednotlivé kroky popsané v kapitole 3 v sekci 3.2. Jelikož je potřeba do samotného překladu více zasahovat a spouštět jednotlivé sekce (`%prep`, `%build`, apod.), nelze zdrojový balíček jádra překládat přímo mockem. Mock však lze použít pro vytvoření izolovaného prostředí pro sestavování jádra z důvodů podobných jako u předešlého nástroje pro sestavování balíčků (viz 5.2). O samotnou konfiguraci a sestavení jádra se může postarat samostatný skript, který bude v chrootu mocku spuštěn pomocí volání `mock --shell`. Před spuštěním tohoto skriptu je nutné ho do chrootu nakopírovat spolu se zdrojovým SRPM jádra pomocí příkazu `mock --copyin`.

Konfigurace

Konfigurace nástroje pro sestavení jádra sestává ze dvou částí. První částí je samotný konfigurační soubor jádra, který může být dodán přímo uživatelem nebo se použije výchozí

nastavení pro danou architekturu z distribučního SRPM.

Nastavení parametrů překladu probíhá podobně jako u předchozího nástroje přes speciální soubory, které obsahují definici maker (viz 5.2.4). Tento soubor může obsahovat nastavení proměnné `%buildid` na zvolený název tak, aby bylo jádro odlišitelné od distribučního jádra (viz 3.2.3). Pokud se nastavení této proměnné nenajde, je vhodné použít nějaký implicitní název, např `.local`. Dále mohou být v souboru maker nastaveny volby `%with` a `%without`. Zapnutím pouze potřebných voleb můžeme sestavování jádra výrazně urychlit.

Příklad nastavení maker pro jádro:

```
%_with_baseonly 1
%buildid .alpha
```

Parametry spuštění se nástroji předávají přes argumenty příkazové řádky. Je tak možné nastavit proměnnou `buildid`, vlastní konfigurační soubor a to, zda se před samotným překladem zavolá konfigurační nástroj menuconfig. Pro tyto možnosti jsem našel inspiraci u nástroje `genkernel`, který slouží ke generování jádra v distribuci Gentoo[5].

Skript spouštěný v chrootu

Parametry pro skript mohou být z volající metody předány jednoduše přes parametry příkazové řádky při jeho spouštění. Následující výpis představuje akce, které by měl skript vykonávat.

- Nainstalování zdrojového SRPM.
- Spuštění přípravné fáze `%prep` k rozbalení SRPM, aplikaci patchů, nastavení proměnných prostředí apod.
- Nastavení proměnné `%buildid`.
- Zkopírování konfigurace pro danou architekturu z přednastavené konfigurace dodávané v distribučním SRPM nebo z konfiguračního souboru dodaného uživatelem.
- Volitelně je zavolán příkaz `make menuconfig` pro interaktivní nastavení konfigurace jádra uživatelem. Takovýto vstup uživatele do procesu sestavování RPM balíčku odporuje principu o neinteraktivitě (viz 3.1.2), a proto je tato volba pouze volitelná. V souborech výsledného RPM balíčku je i konfigurační soubor, který se spolu s dalšími soubory nainstaluje do adresáře `/boot`. Lze tedy zjistit s jakým nastavením bylo jádro sestaveno.
- Spuštění fáze `%build` pro sestavení binárního RPM balíčku.

5.3.2 Třída KernelBuilder

Jak již bylo řečeno, je třída **KernelBuilder** odvozena poděděním od obecnější třídy **Builder**. Následující výpis představuje nové metody a metody, které mají předefinované chování.

mock_install_deps() Tato metoda nainstaluje do chrootu závislosti nutné pro sestavení daného balíčku (v našem případě jádra). Volá se příkaz `mock --installdeps`.

mock_install() Pomocí této metody je možné do chrootu explicitně nainstalovat další balíček. Tato metoda se hodí pro nainstalování balíčku `ncurses-devel`, který je potřebný pro `make menuconfig`.

mock_copy_in() Tato metoda slouží ke kopírování souborů do chrootu mocku.

mock_copy_out() Metoda *mock_copy_out()* zkopíruje soubory z chootu mocku ven do určeného adresáře.

mock_shell() spustí interaktivně příkaz v chrootu. Volá se příkaz `mock --shell`. Interaktivní spuštění znamená, že výstup příkazu se vypisuje na terminál a uživatel může s bežícím programem komunikovat zadáváním vstupu na klávesnici, pokud to spuštěný program umožňuje.

build() Hlavní metoda třídy volá postupně ostatní metody. Inicializuje mock, nainstaluje závislosti nutné pro sestavení jádra, zkopíruje SRPM, skript a případně konfiguraci do chrootu, zavolá skript a výsledek vykopíruje ven z chrootu.

Kapitola 6

Implementace a sestavení distribuce

V této kapitole popíšu zvolený způsob implementace předchozího návrhu. Nejdříve vysvětlím, proč jsem pro implementaci zvolil programovací jazyk Python.

6.1 Jazyk Python

Hlavním důvodem, proč jsem k implementaci zvolil programovací jazyk Python, bylo to, že v tomto jazyku je napsána většina nástrojů, které implementované programy využívají. Zejména u nástroje `yumdownloader` jsem ocenil, že ho v jazyku Python lze importovat jako modul a třídu v něm obsaženou lze použít k vytvoření zděděné třídy.

Jazyk python má i některé další vlastnosti, které spatřuji jako výhody. Při prvním pohledu na kód napsaný v tomto jazyce si nelze nevšimnout přehledného odsazování a absence závorek nebo klíčových slov `begin` a `end` jako oddělovačů bloků kódu. Odsazování se totiž využívá právě k identifikaci bloků, důsledkem je potom lepší čitelnost kódu. Další výhodou je, že se jedná o interpretovaný jazyk. Při testování jednotlivých částí programu tak odpadá mezikrok kompilace. Navíc si lze jednotlivé konstrukce vyzkoušet přímo v interpretu jazyka. Další příjemnou vlastností je automatická správa paměti, o kterou se podobně jako v jazyce Java stará garbage collector.

6.1.1 Styl psaní kódu

Při psaní kódu jsem se snažil dodržovat doporučení PEP-8[8] s výjimkou třídy SourceDownloader, která je napsána odlišným stylem, protože je odvozená z yumu, který dodržuje jiné konvence. V dokumentu PEP-8 se uvádí mnoho doporučení, jak psát kód, aby byl lépe čitelný. Jsou zde doporučení pro odsazování kódu, pojmenování tříd, proměnných a modulů, formátování výrazů, apod.

6.1.2 Pylint

Kód byl průběžně kontrolován nástrojem `pylint(1)`, který dokáže upozornit na syntaktické chyby. Dále tento nástroj pomáhá vynucovat lepší čitelnost kódu tím, že kontroluje odsazování, názvy proměnných, délku řádků, atd. Kontrolovaný kód je ,,oznámkován“ na stupnici od jedné do deseti a výsledná známka je porovnána s uloženou předchozí známkou a lze tedy pozorovat zlepšení nebo zhoršení známky oproti předchozí kontrole. Výsledné průměrné hodnocení kódu je 9.33/10 bez modulu `sourcedownloader`, který vychází z programu `yumdownloader` a používá jiný styl. Se zahrnutím tohoto modulu do testu se průměrné hodnocení sníží na 6.19/10.

6.1.3 Programová dokumentace

Programová dokumentace byla vygenerována pomocí nástroje `epydoc`. Tento nástroj umí vygenerovat dokumentaci z dokumentačních řetězců (docstrings) a z komentářů ve zdrojovém kódu. Komentáře mohou být obohaceny značkami značkovacího jazyka epytext pro popisy parametrů funkcí, instancí proměnných apod. Značkovací jazyk epytext se velmi podobá jazyku, který používá podobný nástroj Javadoc užívaný v prostředí jazyka Java. Výstupem programu epydoc může být programová dokumentace ve formě HTML stránek, které jsou bohatě provázané pomocí odkazů, nebo dokument ve formátu PDF.

6.1.4 RPM balíček

Aby mohla být výsledná sada nástrojů nainstalována běžným způsobem v distribuci Fedora, bylo nutné vytvořit RPM balíček. Výsledný balíček potřebuje ke svému fungování balíčky `createrepo`, `python`, `mock`, `yum`, `yum-utils`, `cpio` a `rpm`. Tyto balíčky jsou zapsány v sekci `Requires` ve spec souboru. Pro otestování správných `Requires` i `BuildRequies` byl balíček sestaven v nástroji `mock`. Nakonec byl program otestován sestavením svého vlastního balíčku.

6.2 Sestavení distribuce

V této sekci je popsán způsob sestavení distribuce z modifikovaných balíčků. Nejdříve jsou představeny různé nástroje pro sestavení distribuce a potom je podrobněji popsáno použití jednoho z nich.

6.2.1 Dostupné nástroje

pungi

Pungi je nástroj pro použití v příkazové řádce, který slouží k vytváření instalačních CD distribuce Fedora. Jako vstup pro nástroj pungi se používá tzv. kickstart soubor, který

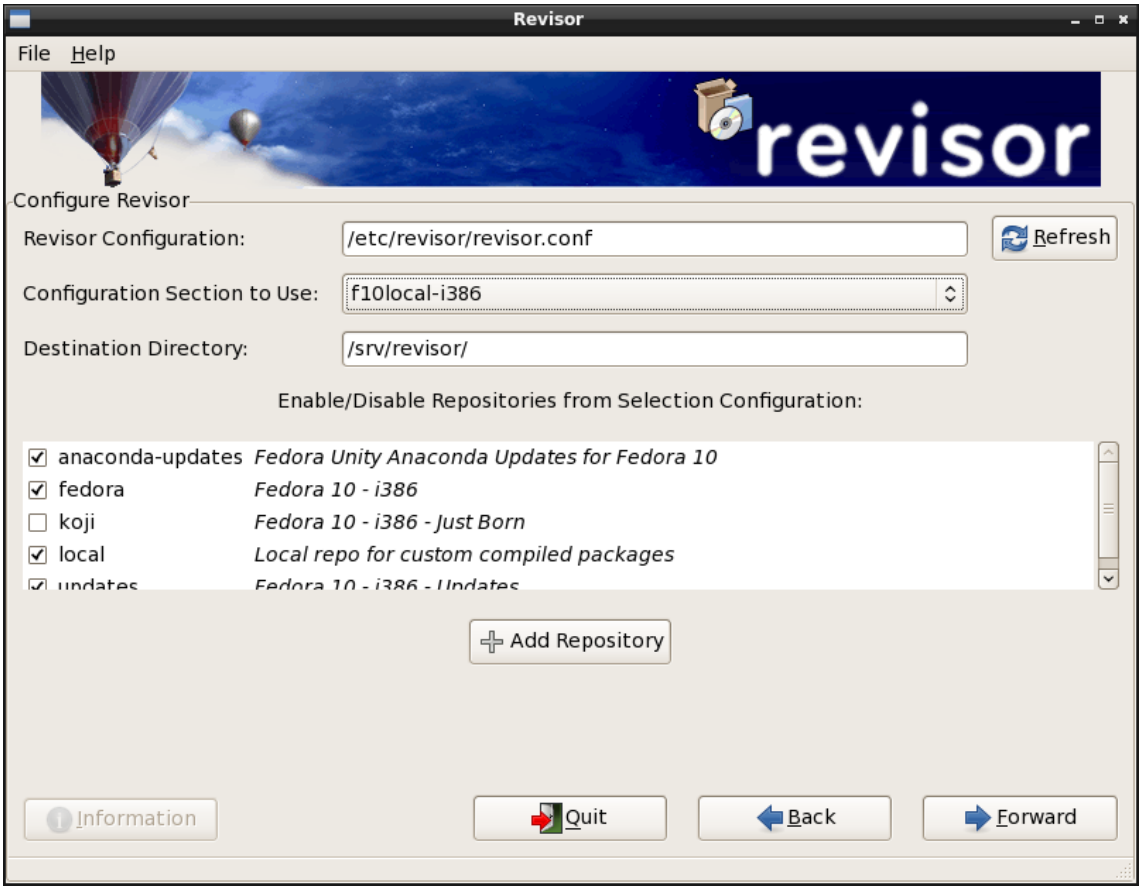
obsahuje seznam balíčků a repozitářů, ze kterých se mají tyto balíčky stáhnout. Výsledkem jsou instalační ISO soubory distribuce.

livecd-tools

Sada skriptů livecd-tools slouží k vytváření ISO obrazů LiveCD distribuce Fedora. LiveCD vytvořená tímto nástrojem mají kořenový adresář přístupný pro zápis, takže je možné na bežící live distribuci instalovat software. Výslednou LiveCD distribuci je také možné nainstalovat na disk.

Revisor

Revisor je grafická nadstavba nad oběma předchozími nástroji. Umožňuje vytvářet klasické instalační ISO obrazy i LiveCD. Nabízí však navíc přehledné grafické rozhraní ve formě průvodce (viz obr. 6.1).



Obrázek 6.1: Grafický průvodce nástroje revisor

6.2.2 Použití programu revisor

Rozhodl jsem se sestavit LiveCD distribuci, protože je vhodnější pro demonstrační účely. Tuto distribuci lze přímo spustit z CD (po vložení do mechaniky, restartu PC a případném nastavení priority zařízení, na kterých se má při startu hledat systém). Tato distribuce může sloužit také k instalaci.

Jako nejvhodnější pro tvorbu distribuce se jeví program `revisor`, neboť umožňuje u použitých repozitářů nastavit jejich prioritu. Program `revisor` dovoluje při sestavování distribuce vycházet z tzv. *kickstart* souboru. Tento soubor předepisuje nastavení repozitářů, seznam balíčků a obsahuje skripty, které se vykonávají při sestavování distribuce. Nastavení voleb v kickstart souboru je jedinou možností jak ovlivnit výsledek práce programů `pungi` nebo `livecd-creator` (z balíčku `livecd-tools`).

Byl vytvořen vlastní kickstart soubor, který vznikl úpravou kickstart souboru pro LiveCD distribuci Fedora s grafickým pracovním prostředím GNOME. Do sekce popisující seznam balíčků byly přidány balíčky vlastní aplikace. Dále byly přidány příkazy, které do výsledného CD zahrnou text této práce, zdrojové kódy vytvořených nástrojů a programovou dokumentaci. Grafický průvodce programu `revisor` našel využití pouze pro nastavení vyšší priority u lokálního repozitáře, aby se balíčky instalovaly přednostně z něj.

Lokální balíčky použité v distribuci byly přeloženy s nastavením voleb kompilátoru `-O2 -march=i686`, aby bylo možné toto LiveCD spustit na širším spektru HW (na architektuře Intel od procesoru Pentium II výše).

Kapitola 7

Závěr

Hlavním přínosem práce bylo vytvoření nástrojů, které slouží k částečné modifikaci distribuce Fedora podle požadavků uživatele.

Jedná se o nástroj `pkgbuild` pro sestavování balíčků ze zdrojových SRPM balíčků, který umožňuje modifikaci voleb kompilátoru a maker, které se používají v průběhu jeho sestavení. Výsledné balíčky jsou umisťovány do lokálního repozitáře, ze kterého je možné je pohodlně instalovat pomocí programu `yum`. Nabízí se však možnost sloučit oba tyto nástroje pro snadnější použití. K tomu byla vytvořena nadstavba `pkginstall` nad nástrojem `pkgbuild`, která dokáže balíčky sestavit a pomocí programu `yum` je nainstalovat nebo aktualizovat při přechodu na novější verzi daného programu.

Dále byl vytvořen nástroj `buildkernel` pro sestavení jádra Linux podle uživatelem zadané konfigurace. Tento nástroj sestavuje jádro z distribučního SRPM balíčku. Vychází při tom z konfigurace distribučního jádra a je na uživateli, jak si tuto konfiguraci upraví. Má při tom možnost dodat nástroji vlastní konfiguraci nebo si ji vygenerovat během volání nástroje.

Implementace byla provedena v programovacím jazyku Python, který je dle mého názoru vhodný na takovéto projekty vzhledem k velkému množství dostupných funkcí v podobě modulů, které jsou s ním standardně dodávány. Je v něm také napsána většina programů, které implementované nástroje využívají. Není tak potřeba do systému instalovat podporu pro další programovací jazyk.

Na tento projekt je možné navázat rozšířením funkčnosti jednotlivých nástrojů. Pro nástroj k sestavování SRPM balíčků by bylo vhodné navrhnout sofistikovanější řešení závislostí, které by mohlo brát v úvahu i závislosti na soubory, které se objeví až ve výsledných binárních RPM balíčcích apod. Nástroj pro sestavení jádra by se dal rozšířit například o automatickou detekci hardware, na kterém tento nástroj běží a cílem by tak mohlo být sestavení minimálního jádra pouze s nezbytnými ovladači a podsystémy.

Literatura

[1] FedoraWiki: Co je Fedora Linux. Příručka uživatele Fedora 10, 2008, [cit. 2008-12-29].
URL: http://wiki.fedora.cz/doku.php?id=fcz:o_fedore#co_je_fedora_linux

[2] Foster-Johnson, E.: RPM Guide. el. kniha, 2005.
URL: http://docs.fedoraproject.org/drafts/rpm-guide-en

[3] Gentoo Foundation: About Gentoo. Oficiální webové stránky projektu Gentoo, 2009, [cit. 2009-04-26].
URL: http://www.gentoo.org/main/en/about.xml

[4] Gentoo Foundation: Gentoo Handbook. Instalační příručka, 2009, [cit. 2009-04-24].
URL: http://www.gentoo.org/doc/en/handbook/handbook-x86.xml

[5] Gentoo Foundation: Gentoo Linux Genkernel Guide. Příručka nástroje genkernel, 2009, [cit. 2009-05-18].
URL: http://www.gentoo.org/doc/en/genkernel.xml

[6] Nečas, D.: Rukověť baliče RPM. Seriál článků na AbcLinuxu.cz, 2005, [cit. 2009-04-26].
URL: http://www.abclinuxu.cz/clanky/navody/rukovet-balice-rpm-i-uvod

[7] Python Software Foundation: Python v2.7a0 documentation. Dokumentace jazyka Python, 2009, [cit. 2009-05-16].
URL: http://docs.python.org/dev/library/configparser.html

[8] Rossum, van, G.; Warsaw, B.: PEP8 – Style Guide for Python Code. Oficiální webové stránky programovacího jazyka Python, 2009, [rev. 2009-01-22].
URL: http://www.python.org/dev/peps/pep-0008/

[9] The Fedora Project: Legacy/Mock. Wiki stránky projektu Fedora, 2008, [cit. 2009-05-13].
URL: https://fedoraproject.org/wiki/Legacy/Mock

[10] The Fedora Project: Docs/CustomKernel. Wiki stránky projektu Fedora, 2009, [cit. 2009-01-04].
URL: http://fedoraproject.org/wiki/Docs/CustomKernel

[11] The Fedora Project: Packaging:Guidelines. Wiki stránky projektu Fedora, 2009, [cit. 2009-04-26].
URL: https://fedoraproject.org/wiki/Packaging:Guidelines

[12] The Fedora Project: Projects/Mock. Wiki stránky projektu Fedora, 2009, [cit. 2009-04-29].
URL: https://fedoraproject.org/wiki/Projects/Mock

Seznam příloh

A Uživatelská příručka

B Datový nosič CD s výslednou distribucí, programovou dokumentací a zdrojovými texty práce

Příloha A

Uživatelská příručka

A.1 Popis nástrojů

Pkgbuild je sada nástrojů pro usnadnění kompilace zdrojových SRPM balíčků. Obsahuje nástroj pro sestavování balíčků, yum wrapper pro instalaci a aktualizace a nástroj pro vlastní sestavení jádra Linux.

Obsahuje tyto tři nástroje:

- `pkgbuild`
- `pkginstall`
- `buildkernel`

A.1.1 pkgbuild

Nástroj `pkgbuild` stáhne SRPM z nastavených repozitářů a sestaví je v mock chrootu. Pro jednotlivé balíčky může být specifikováno speciální nastavení maker a voleb pro rpmbuild. Tato nastavení se nacházejí v adresáři `/etc/pkgbuild`.

Způsob použití: `pkgbuild název_balíčku`

A.1.2 pkginstall

Nástroj `pkginstall` spojuje `pkgbuild` a `yum`. Argumenty programu jsou stejné jako u nástroje `yum` s tím rozdílem, že balíčky jsou nejdříve sestaveny a potom nainstalovány z lokálního repozitáře.

Způsob použití: `pkginstall [nastavení] příkaz`

Seznam příkazů:

install Sestaví a nainstaluje balíček nebo více balíčků.

reinstall Opětovné sestavení a instalace balíčku nebo více balíčků.

update Aktualizace balíčku nebo balíčků, které mají k dispozici novější SRPM.

Nastavení a další příkazy jsou přímo předány programu `yum`. Pro nápovědu k programu `yum` spusťte `yum --help`.

A.1.3 buildkernel

Nástroj `buildkernel` slouží k vlastnímu sestavení jádra Linux.

Způsob použití: `buildkernel [volby] SRPM`

-h, –help zobrazí nápovědu.

-m, –menuconfig zavolá menuconfig před sestavením jádra.

-b BUILDID, –buildid=BUILDID nastaví identifikátor jádra, který slouží k rozlišení mezi distribučním jádrem a jádrem sestaveným uživatelem.

-c CONFIG, –config=CONFIG použije zadaný konfigurační soubor.

A.2 Nastavení nástrojů

A.2.1 Hlavní nastavení

Všechny tři nástroje přistupují k nastavení ve společném souboru. Tento soubor je implicitně nainstalován jako `/etc/pkgbuild/config` a měl by obsahovat rozumné nastavení, které většinou není potřeba měnit.

A.2.2 Makra

V adresáři `/etc/pkgbuild/macros` může být provedeno předefinování maker pro sestavované balíčky. Nachází se zde soubory pojmenované podle SRPM balíčků obsahující nastavení pro tyto balíčky. Jestliže není nalezen konfigurační soubor pro právě sestavovaný balíček, použije se soubor `_default`.

Například soubor `/etc/pkgbuild/macros/kernel` může obsahovat:

```
%_with_baseonly 1
%_with_firmware 1
%buildid .local
```

A.2.3 rpmrc

Nastavení, která můžou být za normálních okolností specifikována v `rpmrc` souboru, mohou být, podobně jako u maker, provedena pro jednotlivé balíčky. Konfigurační soubory se nacházejí v adresáři `/etc/pkgbuild/rpmrc`.

Například soubor `/etc/pkgbuild/rpmrc/_default` může obsahovat:

```
optflags: i386 -O2 -march=native
```