ZÁPADOČESKÁ
UNIVERZITA
V PLZNI

Katedra informatiky a výpočetní techniky
Operační systémy

Správce virtuálních strojů

mail: novotny@students.zcu.cz

Jiří Novotný A09N0032P
Zdeněk Janda A09N0076P
Miroslav Hauser A09N0037P

1 Zadání

Navrhněte a implementujte model správce virtuálních strojů. Shell tohoto správce bude obsahovat tyto příkazy: *cat, cd, echo, exit, kill, kshell, ls, man, ps, pwd, shutdown, sort*. Shell bude dále uchovávat aktuální pracovní adresář a historii použitých příkazů. K ovládání správce použijte vhodné rozhraní uživatel/stroj.
Doporučený postup:

1. Napište si terminálové okno.
2. Navrhněte gramatiku pro shell.
3. Navrhněte a implementujte syntaktický analyzátor.
4. Navrhněte strukturu správce virtuálních strojů.
5. Udělejte propojení na souborový systém, napište váš cat a ukažte jeho funkčnost.
6. Doplňte model o vykonávání příkazů shellu v hierarchii virtuálních strojů - init, login, shell, program xyz, ...
7. Přidejte přesměrování a roury.
8. Napište další programy pro příkazy shellu, logout, shell, date, ...
9. Model můžete rozšířit o send a receive mezi uživateli, vykonávání v pozadí, příkazy bg a fg.
10. K historii příkazů uchovávejte i adresáře, ve kterých byly použity
11. Dokončete detaily.
12. Testujte.

Během práce dodržujte tyto konvence:

- identifikátory a komentáře se píší anglicky
- názvy package pouze malými písmeny
- class soubory se generují do jiného adresáře (ne k zdrojákům)
- místo třídy kolekce se používá rozhraní

2 Popis implementace

2.1 Konzole

Konzole se vytváří jako objekt třídy *UserInterface*, která je oděděna od třídy *JFrame*. Tento frame obsahuje *JTextAreu* jako jediný prvek. Jedná se o základní vstup a výstup správce virtuálních strojů. V rámci konstruktoru jsou nastaveny globální proměnné, vzhled okna konzole, namapování akcí a jako poslední je spuštěn první shell. Globální proměnné představují již zmíněnou *JTextAreu*, dále hodnotu offsetu od začátku textu *TAOff* a *lineHead*, což je univerzální začátek každého řádku konzole tzv. line header. Namapování jednotlivých akcí je nejobsáhlejší část vytváření konzole a obsahuje jak reakce na klávesnici, tak na myš. Podrobnému popisu se věnuje podkapitola o mapování akcí viz dále. Spuštění shellu obstarává metoda *createShell()*, která bude rozebrána v další části dokumentace.

2.1.1 Reakce na klávesnici

K namapování akcí klávesnice je použita metoda *consoleKeyActions(keyEvt)*. Ta v sobě obsahuje switch na jednotlivé kódy stisknutých kláves (*getKeyCode()*). Vyhodnocované klávesy jsou: *ENTER, LEFT, RIGHT, UP, DOWN, PAGE UP, PAGE DOWN, HOME, TAB a BACKSPACE*. Klávesa *ENTER* umožňuje zpracování aktuální řádky zadané do konzole ať už shellem nebo právě běžící aplikací. Šipky slouží k práci s historii příkazů a pohybu v zadaném textu. Klávesa *HOME* je pozměněna tak, aby v případě běhu shellu skákala pouze na konec vypsaného textu (za line header), tedy na hodnotu *TAOff*. V ostatních případech skáče až na začátek aktuálního řádku. Klávesy *PAGE UP* a *PAGE DOWN* jsou zakázány úplně, aby nebylo možné skočit do již zpracované oblasti konzole. Klávesa *TAB* doplňuje rozepsaný příkaz resp. cestu v zadávaném řádku a klávesa *BACKSPACE* má opět přidáno omezení, aby nebylo možné mazat zpracované řádky konzole. Poslední akcí klávesnice je tzv. KeyStroke, tedy kombinace více kláves (klávesová zkratka). Je implementována reakce pouze **CTRL+D** určená k zaslání signálu odebrání konzole a ukončení aktuálního procesu.

2.1.2 Reakce na myš

Kliknutí nebo označení textu v konzoli - Pokud se uživatel pokusí kliknout jinam než na konec konzole (aktuální řádek) nebo v případě označení textu je potřeba zamezit změně pozice kurzoru a tím i možnosti psaní mimo aktuální řádek. Oba problémy jsou řešeny metodou *mouseReleased(MouseEvent evt)*. Tedy kdykoli je v konzoli uvolněno tlačítko myší (ať po kliknutí nebo po

označení) je kurzor nastaven na aktuální řádek (na konec textu konzole, který dán hodnotou offsetu *TAOff*).

2.2 Gramatika

2.2.1 Návrh

Návrh gramatiky byl odvozen od LINUXu. Pro zápis návrhu jsou použity tyto regulární výrazy:

- 'x' - znak x
- (x)? - žádný nebo jeden výskyt
- (x)* - žádný nebo několik výskytů
- xyz - pravidlo
- XYZ - lexikální výraz

Následně bude návrh gramatiky vypadat takto:

```
parse   ->  line (bg)?
line    ->  first ('<' in)? (next)* ('>' out)?
first   ->  cmd
next    ->  '|' cmd
cmd     ->  par (args)*
args    ->  par
in      ->  par
out     ->  par
bg      ->  '&'
par     ->  CHAR | STRING | ichar
ichar   ->  ICHAR
```

Na první pohled se může zdát návrh gramatiky zbytečně složitý, ale takto detailní rozložení je dále použito při návrhu lexikálního resp. syntaktického analyzéru. Oba analyzéry jsou generovány pomocí nástroje ANTLR 3.2, který podle návrhu gramatiky vytvoří kód v Javě. Ten je navíc doplněn obslužným kódem pro jednotlivá pravidla tak, aby práci se zpracovanou řádkou co nejvíce ulehčil (proto je návrh tak podrobný). Celý návrh gramatiky s plnými pravidly pro lexikální i syntaktický analyzátor lze nalézt v souboru s gramatikou (OSVM_grammar.g).

2.2.2 Lexikální analyzér

Pro lexikální analyzér jsou definovány základní řídící znaky (roura, přesměrování, ad.) a množina povolených symbolů pro názvy příkazů resp. souborů (CHAR, STRING). Dále je definován i doplněk obou množin ICHAR. Veškeré bíle znaky a znaky z množiny ICHAR jsou přeskakovány.
Použité množiny:

- CHAR
  ('a'..'z' |'A'..'Z' |'0'..'9' |'/' |'_' |'-' |'?' |'.' |'..' |':')
- STRING
  CHAR*
- ICHAR
  !(' ' |'|' |'<' |'>' |'\n' |'&')

2.2.3 Syntaktický analyzér

Syntaktický analyzér rozdělí řádek do struktury *ArrayList*, který v sobě obsahuje další *ArrayList<String>*. V něm jsou uloženy jednotlivé argumenty příkazu a jako poslední položka je vždy název příkazu. Navíc ukládá cestu k vstupnímu resp. výstupnímu souboru do String proměnné *in* resp. *out* a poskytuje k nim přístup pomocí metod *getIn()* a *getOut()*. V poslední řadě nastavuje vlajky *bg* (pokud je příkaz spuštěn na pozadí) a *invalid* (pokud řádka obsahuje nějaký symbol z množiny ICHAR). Ty jsou po zpracování řádky přístupné přes metody *isBackgrounded()* a *containsInvalid()*.

2.3 Struktura VMM

Hlavní myšlenka tohoto software byla převzata z GNU/Linux. Z důvodu absence zavaděče jádra a dalších procedur, nutných ke startu "běžného" operačního systému, jsme si však mohli dopřát několik zjednodušení.
Pro zavedení jádra operačního systému jsme využili jednoduše metodu *void main()*. Takže ihned po spuštění programu se nám zavede jádro. Jádro spustí (vytvoří instance) potřebné manažery. V našem případě jsou to, manažer uživatelů (*UserManager.java*), manažer uživatelských rozhraní (*UIManager.java*) a manažer nastavení (*PropertyManager.java*).
Po zavedení managerů je spuštěn *INIT* proces. PID (Process IDentifier) je vždy rovno 1. Zde je vidět analogie s GNU/Linux. V operačním systému Linux se tento proces (*INIT*), nastavený jako daemon process, používá k startování jiných procesů jádra. *INIT* proces je kořenem stromu procesů, který za běhu OS (Operating System) vytváří. V naší implementaci je *INIT*

pouze prázdný process, který nic nedělá. Ovšem jeho implementace umožňuje použití ke stejnému účelu jako v Linuxu.
Po zavedení a spuštění *INIT* procesu je spuštěno přihlašovací okno. Zde je umožněno přihlášení uživatelů. Naše implementace neobsahuje ověření heslem. Po zadání uživatelského jména získá uživatel terminál, pomocí kterého lze komunikovat se systémem.

2.4 Vykonávání příkazů

Vykonávání příkazů je rozděleno do několika částí. První je zpracování řádky v konzoli, to se provádí po potvrzení napsané řádky klávesou ENTER. Nejprve je metodou stdReadLn() načten z konzole aktuální napsaný řádek. Pokud se jedná o příkaz konzole (*clr*), pak se provede ihned. V ostatních případech se příkaz předá aktuálně spuštěnému procesu. Toto je druhá část zpracování o kterou se stará metoda *processLine()*, ta je abstraktní a musí ji tedy obsahovat všechny procesy. Implementována je ovšem jen u procesů, které se aktivně mohou dostat ke konzoli. V našem případě pouze cat, sort a kshell. První dva příkazy pouze čtou z konzole a zpracování takto načtených dat odpovídá popisu procesů (vypsání resp. řazení).

2.4.1 kshell

kshell se zpracovanou řádkou dále pracuje. Nejprve kontroluj zda není řádka prázdná a zda pokud obsahuje příkazy *kshell* nebo *exit*, tak neobsahuje nic dalšího. Po provedení těchto kontrol se pomocí generovaného parseru řádka rozdělí na jednotlivé příkazy (včetně argumentů a přesměrování). Pokud řádka obsahovala nějaké neplatné symboly (z množiny ICHAR viz výše), pak je uživatel upozorněn a znaky jsou přeskočeny, ale zpracování pokračuje dál předáním potřebných informací metodě *createProcess()*.

2.5 Přesměrování a roury

2.5.1 Roury

Na implementaci rour byly využity Javovské třídy *PipedWriter* a *PipedReader*. Použití těchto tříd je jednoduché a spolehlivé řešení. Instanci třídy *PipedReader* přiřadíme instanci třídy *PipedWriter*. Tyto instance jsou pak přiřazeny k příslušnému *BufferedReaderu/Writeru*, který je používán vlákny pro čtení z roury respektive pro zápis do roury.

2.5.2 Propojení na souborový systém

Aby se implementace souborového systém co nejvíce podobala modelu rour, byly vytvořeny třídy *KSHWriter* a *KSHReader*, které jsou oddděné od tříd *Writer* a *Reader* stejně jako *PipedWriter/Reader*. Oddědění vyžaduje implementaci základních metod. Ta byla provedena pomocí tříd *FileWriter/Reader*. K vlastnímu čtení a zápisu jsou pak využity třídy *BufferedWriter/Reader* stejně jako u rour. Pomocí nich byli vytvořeny metody pro otevření/zavření souboru a základní metody pro čtení a zápis souborů.

2.5.3 Standardní vstup a výstup

Vytvořením *StdOut* a *StdIn* interfaců bylo procesům umožněno využívat souborového i konsolového vstupu/výstupu bez nutnosti používat rozdílné metody. Rozhraní obsahují metody pro otevření a zavření vstupu nebo výstupu a základní metody pro čtení a zápis. Konsolový user interface a roura implementují obě tato rozhraní. *KSHWriter* a *KSHReader* implementují jim příslušná rozhraní. K standardnímu vstupu a výstupu byl ještě doplněn interface pro chybový výstup.

2.6 Příkazy

Každý příkaz je reprezentován svou třídou vyjma příkazů *exit* (příkaz shellu) a *clr* (příkaz konzole). Tyto příkazy jsou odděděny od naší třídy Process, proto obsahují metodu pro zpracování řádky a signálu, výchozí metodu pro spuštění a případně metody pro zpřehlednění kódu. Na začátku všech spustitelných příkazů je provedena kontrola parametrů. Zde se v našem případě kontroluje pouze *-h* (u *ps* ještě *-u*). Poté už následuje funkční kód definovaný zadáním.

3 Uživatelská příručka

3.1 Překlad

Program lze snadno přeložit pomocí:

```
X:\app_path\>compile.bat
```

a spustit:

```
X:\app_path\>run.bat
```

3.2 Ovládání programu

Ovládání správce je intuitivní. Po spuštění (viz předchozí část) se objeví okno přihlášení. Po zadání uživatelského jména (systémem akceptovaná uživatelská jména: hauz, k4chn1k, sysek, guest) a potvrzení dojde k přihlášení uživatele k virtuálnímu stroji. Zde je možné používat základní příkazy shellu, včetně jejich argumentů a přesměrování vstupů resp. výstupů. Implementované příkazy: *cat, cd, echo, exit, kill, kshell, ls, man, ps, pwd, shutdown, sort.* Obecná syntaxe:

```
cmd1 [file]* [< ifile] [| cmd\_next]* [> ofile]
```

Výjimkou oproti zadaní je příkaz *ps*. Ten lze použít s přepínačem *-u*, čímž se zobrazí pouze uživatelovy procesy. Dále byl přidán příkaz *clr*, který smaže obsah konzole.

4 Závěr

Zadání semestrální práce bylo splněno v povinném rozsahu. Navíc byl přidán přepínač *-u* u příkazu *ps* a příkaz konzole *clr*.

4.1 Jiří Novotný

Vytvořil jsem tyto části kódu:

- Gramatika
- Odchytávání akcí v konzoli
- KSHell a všechny externí příkazy

Možná vylepšení:

- Vylepšení klávesy TAB na celý příkaz
- Další parametry příkazů

Hlubší poznání Javy, ke kterému jsem se během studia jako "hardwarář" nedostal, a práce s nástrojem ANTLR. Navíc možnost zkusit si práci v týmu. Zadání bylo zajímavé, bez výhrad.

4.2 Zdeněk Janda

Mým hlavním úkolem v týmové práci bylo vytvoření IO balíku.
Díky výpomoci na dalších různých částech projektu jsem se věrně seznámil s celou aplikací a řešil mnoho zajímavých situací. To vyžadovalo dobrou komunikaci se zbytkem týmu. Díky práci jsem se blíže seznámil s verzovacím systémem a mnohokrát ocenil jeho výhody. Práce pro mě byla velkým přínosem díky nabitým vědomostem a zkušenostem.

4.3 Miroslav Hauser

Práce byla velkým přínosem. Studium by mělo obsahovat více takovýchto týmových projektů. Dále nám práce umožnila nahlédnout, alespoň trošku, pod kapotu současných operačních systémů. Tyto zkušenosti jsou k nezaplacení.

Dokumentace byla vygenerována pomocí nástroje TEX.