Západočeská univerzita v plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Semestrální práce
KIV/OS
Simulace operačního systému

Plzeň, 2012

Martin Bydžovský
A12N0058P

Jakub Daněk
A12N0059P

Jiří Zikmund
A12N0103P

Zadání

Simulátor operačního systému. V Javě naprogramovat aplikaci, která bude napodobovat chování operačního systému – správa procesů, souborový systém, IO zařízení, služby OS, přerušení… Součástí práce je třeba naprogramovat několik ukázkových procesů, které budou demonstrovat správnost a funkčnost systému.

Technická dokumentace

Proces a jeho průběh

Každý proces implementovaný v našem simulovaném OS, musí být potomkem třídy `Process`. Tato abstraktní třída uchovává společné informace pro všechny procesy (běhové vlákno, vstupní/výstupní streamy, zpracování signálů…). Dále obsahuje metody pro zjišťování aktuálního stavu procesu a jeho ovládání.

Také funguje jako wrapper nad metodou `run`, kterou oddděné procesy přetěžují a ve které vykonávají svoji činnost. Odchytává výjimky vyhozené z této metody, nastavuje příznaky (ne)úspěšného skončení a informuje o této skutečnosti `ProcessManager`.

O vytvoření instance nového procesu se stará služba operačního systému `createProcess`, která toto volání deleguje na objekt `ProcessManager`. Jeho hlavní funkcí je uchovávání informací o běžících procesech, jejich stavu a spouštění nových procesů. Získání reference na instanci konkrétního procesu potom provádí za použití Java Reflection API.

Poté co proces skončí, je notifikován `ProcessManager` (na základě observer pattern), který určí, kdo bude dále na popředí. V tuto chvíli má skončený proces pořád záznam v tabulce procesů. Ten je odebrán až poté, co jiný proces zavolá službu OS `readProcessExitCode` a zjistí, návratovou hodnotu procesu. Po tomto volání je proveden závěrečný úklid po procesu a je definitivně odstraněn ze systému.

Vstupní parser a gramatika shellu

O analýzu vstupního příkazu se stará třída `InputParser`. Jedním průchodem zpracuje vstupní řetězec a vytvoří z něj spojovou strukturu `ParseResult`, která obsahuje informace o nalezených argumentech, přesměrovaném vstupu/výstupu (případně jestli se má soubor appendovat nebo ne), flag jestli se příkaz spouští na pozadí a referenci na další příkaz `ParseResult` (je to spojová struktura.

Shell tedy ve svém běhu přečte řádek vstupního textu, za pomoci třídy InputParser zjistí, jaké příkazy se mají provést, a voláním služeb OS tyto příkazy provede.

Parser vstupního textu rozpoznává následující gramatiku:

```
COMMAND := PROCESS PARAMETER PIPE ASYNC
PROCESS: = string
PIPE := e | „ | PROCESS PARAMETER PIPE“
ASYNC := e | &
PARAMETER := e | >string | >>string | <string | string
             |PARAMETER PARAMETER
```

Jak je vidět, nerozlišuje se rozdíl mezi parametrem a argumentem (jak bylo ukazováno na cvičení) a navíc přesměrování vstupů/výstupů lze kombinovat s parametry nezávisle na pořadí – ostatně takto funguje i linuxový bash. Následující příkaz je tedy zcela validní:

```
p1 param1 >>out.txt param2 –arg1 | p2 > err param1 param2 < input
```

Abstrakce V/V

Veškeré vstupně-výstupní operace jsou prováděny pomocí implementací rozhraní `IDevice` a jeho potomků. Rozhraní jsou navržena tak, aby jednotlivé procesy byly funkčně zcela nezávislé na tom, zda jejich vstup či výstup je soubor, roura nebo např. terminálové okno.

Základní implementace využívá vstupních a výstupních proudů v Javě, respektive jejich nadstaveb `BufferedReader` a `BufferedWriter`. Roury a propojení s terminálem jsou pak realizovány frontou. Operace čtení i zápisu z/do roury/terminálu jsou synchronizovány – zároveň smí k zařízení přistoupit nejvýše jeden proces. V případě, že je zařízení používáno, proces se zablokuje dokud není zámek uvolněn. Samotná operace čtení je blokující a operace zápisu je neblokující.

Čtení ani zápis z/do obecného proudu (např. souboru) není synchronizován a v případě nutnosti musí být synchronní přístup k zařízení ošetřen v rámci uživatelských procesů.

Souborový systém

Souborový systém je inspirován unixovými systémy. Jedná se jeden strom s kořenovým adresářem „`/`“. Ten je vytvořen v hostitelském systému na cestě `$HOME/os` (resp. na alternativě v systémech Windows). Implementace neumožňuje opustit v rámci simulace tuto složku.

Soubory či složky začínající znakem „**.**“ (tečka), jsou považovány za skryté.

Je možné vytvářet soubory a složky a číst a zapisovat z/do souboru. Mazání není z bezpečnostních důvodů povoleno.

Kompatibilita s Windows

Jelikož systém Windows neumožňuje zakládat soubory či složky obsahující určité znaky, je u všech cest prováděna kontrola. V případě nálezu některého ze znaků uvedených níže je vrácena výjimka. Implementované procesy se z této výjimky zotaví a zobrazí uživateli chybovou hlášku.

Zakázané znaky:
`\, *, ?, :, ", <, >, |`

Uživatelský manuál

Přihlášení

Po startu systému budete vyzváni k zadání uživatelského jména. Zadejte libovolné jméno a stiskněte enter. Budete přihlášeni a aktuální adresář bude nastaven na `/home/<uživatelské jméno>`. Pokud neexistuje (např. je to první přihlášení uživatele), bude příslušná složka vytvořena.

Práce v systému

Po přihlášení můžete zadávat jednotlivé příkazy. Přístupné jsou funkce přesměrování vstupu a výstupu z/do souboru či jiného procesu.

Např.

```
ls -a | wc > pocet_souboru.txt
```

spočítá všechny adresáře a soubory v aktuálním adresáři, včetně skrytých, a výsledek uloží do souboru `počet_souboru.txt`.

Úlohu je možné spustit na pozadí zakončením příkazu znakem *&*. Vše za tímto znakem na příkazové řádce se ignoruje!

Příkazy

Pro zobrazení nápovědy k příkazu napište:

```
man <název_příkazu>
```

Následuje seznam dostupných příkazů, popsány jsou pouze rozdíly proti standardnímu chování v bashi.

- cat
- echo
- echopid – opakovaně vypíše svůj PID
- kill – pošle SIGKILL zadanému procesu (dle PID)
- login – přihlášení jako jiný uživatel (ekvivalent *su* v linuxu)
- ls
- man
- mkdir – vytváří celou adresářovou strukturu
    - *mkdir /neexistujici1_adr/neexistujici_adr2* je platný příkaz
- ps
- pwd
- shell – spustí novou instanci příkazového interpretu (ekvivalent k např. *bash* v linuxu)
- shutdown – ukončí všechny procesy a následně celou simulaci OS
- sleep

- sort
- touch
- wc
- cd
- exit

Ukončení systému

Systém lze ukončit dvěma způsoby:

1. příkazem `shutdown`
2. křížkem terminálového okna

Při ukončení jsou zastaveny všechny procesy a uvolněny všechny zdroje (např. soubory). Nakonec je zavřeno terminálové okno a simulace končí.

Závěr

Toto byla zatím asi nejrozsáhlejší semestrální práce, které jsme se na univerzitě věnovali. Proto už od začátku jsme měli na paměti, že je třeba aplikaci důsledně navrhnout, a veškeré změny a rozšiřování provádět s dostatečným nadhledem tak, aby do sebe vše komplexně zapadalo.

Práci jsme měli rozdělenou následovně:

- Martin Bydžovský se staral o implementaci `ProcessManageru`, terminálového okna, vstupního parseru a částečně o tvorbu bázového procesu.
- Jakub Daněk zajistil `FileManager` a veškeré IO zařízení (soubor, klávesnice, roura…) a také se postaral o kompletní synchronizaci jádra a jeho služeb.
- Jiří Zikmund vytvořil všechny procesy systému a částečně se podílel na tvorbě bázového procesu.

Aplikace splňuje všechny povinné body zadání, máme dokonce nějaké procesy navíc – touch, wc, login, sleep. Další zajímavostí, kterou jsme vytvořili je naprosto uzavřený filesystém v rámci simulovaného OS – všechny souborové operace je možné provádět pouze v něm. Dále máme implementované víceúlohové prostředí, tzn. spouštění procesů na pozadí a následné posílání signálů jednotlivým procesům.

V neposlední řadě silnou stránkou naší aplikace je modulární skladba systému: Celkem jednoduché jádro, které rozděluje práci mezi své managery – `ProcessManager`, `SignalDispatcher`, `FileManager` atd… Tato skladba jednoduše umožňuje další rozšiřování aplikace.

Mezi slabší stránky naší aplikace patří jednak parser, který nerozpoznává vše tak, jako bash – např. otevírání file deskriptorů (4 <> file) nebo jsme neimplementovali operátor &&. Zde bych rád podotknul věc, která byla explicitně zmíněná v zadání – a sice že u operátorů přesměrování vstupu/výstupu nemusí být mezera. Toto mi připadá krajně nevhodné (bash v tomto ohledu bohužel nefunguje úplně 100% podle očekávání). Na problém narazíme např. při příkazu:

```
$ cat soubor>out2>err
```

Znamená to „přesměruj stdOut do out a stdErr do err“ nebo „přesměruj stdOut do out2 a pak znovu stdOut do err „?

Dále nás mrzí, že jsme nestihli implementovat přepínání procesů z popředí na pozadí a zpět. Nicméně myslíme si, že ani další týden navíc by nám nedovolil toto perfektně odladit a zprovoznit.

Několikrát jsme narazili na limity virtuálního stroje JVM – například násilné okamžité ukončení procesu (resp. vlákna) není úplně možné bez rizika, že by některý objekt (typicky zařízení) nezůstal v nekonzistentním stavu. Procesy tedy musí být „hodné“ a v případě delšího výpočtu pravidelně kontrolovat flag, zda mají skončit a podle toho zareagovat.

Celkově semestrální práci hodnotíme velice kladně, a přestože patřila mezi náročnější a stane se z ní trezorová záležitost, práce na ní nás opravdu bavila a byla celkové velmi přínosná.