Semestrální práce z předmětu KIV/OS

Simulace operačního systému

Eliška Mourycová

A20N0061P

Ondřej Drtina

A20N0077P

Stanislav Král

A20N0091P

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Obsah

1 Zadání

Zadáním semestrální práce byla simulace operačního systému. Úkolem bylo navrhnout a s využitím připravené kostry simulátoru v jazyce C++ implementovat aplikaci, která bude simulovat chování operačního systému.

Pro simulaci bylo potřeba navrhnout a implementovat správu procesů, souborový systém a konkrétní uživatelské příkazy a programy.

1.1 Požadované programy

Seznam a stručný popis požadovaných uživatelských programů je uveden zde (pro podrobnější popis jednotlivých příkazů viz podsekci 2.2):

Příkaz	**Význam**
`echo`	Vypíše řetězec zadaný v argumentu
`cd`	Změní pracovní adresář aktuálního shellu
`dir`	Vypíše položky, které se nachází v zadaném adresáři
`md`	Vytvoří nový adresář
`rd`	Smaže zadaný adresář
`type`	Vypíše obsah zadaného souboru
`find`	Vypíše počet řádek zadaného souboru
`sort`	Seřadí jednotlivé řádky zadaného souboru
`tasklist`	Vypíše seznam běžících procesů
`shutdown`	Ukončí všechny procesy
`rgen`	Začne vypisovat náhodně vygenerovaná čísla v plovoucí čárce
`freq`	Sestaví frekvenční tabulku bytů, kterou pak vypíše pro všechny byty s frekvencí větší než 0
`shell`	Spustí nový shell

Tabulka 1: Požadované příkazy shellu

2 Popis hlavních modulů

2.1 User space

User space je část operačního systému, která pomocí systémových volání žádá kernel o služby pro obsluhu uživatelských programů. Tato logika je použita i v naší práci. Funkcionalita uživatelských programů (jejich kód, spouštění, apod.) je implementovaná v projektu `user`. Jejich spouštění a případná orchestrace do pipeline je řešena v `shell.cpp`.

2.2 Podrobný popis podporovaných příkazů

Zde je podrobnější popis podporovaných příkazů pro náš simulátor. Implementace funkcionality s nachází v příslušných `.cpp` zdrojových souborech (tj. `nazev_prikazu.cpp`).

`echo`

`echo` jako argument očekává řetězec, který má vypsat na standardní výstup. Pokud se v řetězci nachází 'speciální' znaky (tj. znak | nebo < nebo >), je potřeba řetězec uzavřít do uvozovek ("), pokud výstup programu echo chceme přesměrovat na vstup jiného programu nebo do souboru. Příklad:

```
> echo "hello" | freq
  0xa : 1
  0x65 : 1
  0x68 : 1
  0x6c : 2
  0x6f : 1 /* vystup programu freq */
> echo hello | freq
  hello | freq /* vystup programu echo */
```

Ukázka kódu 1: Ukázka chování programu echo

`cd`

`cd` změní pracovní adresář aktuálního shellu. Jako argument očekává cestu (relativní nebo absolutní) k novému adresáři, který má shellu nastavit jako pracovní. Pokud argument není zadaný, pracovní adresář se nemění.

`dir`

`dir` na standardní výstup vypíše položky uložené v adresáři. Cesta k požadovanému adresáři se zadává jako argument příkazu. Pokud argument není zadaný, vypíše se obsah pracovního adresáře.

`md`

`md` vytvoří nový adresář. Cesta k novému adresáři se zadává jako argument příkazu. TODO

`rd`

`rd` smaže adresář zadaný v argumentu. TODO

`type`

`type` na standardní výstup vypíše obsah zadaného souboru. Cesta k požadovanému souboru se zadává jako argument příkazu. Pokud argument není zadaný, `type` začne číst ze standardního vstupu, dokud nepřečte znak `EOT`.

`find`

`find` na standardní výstup vypíše počet řádek zadaného souboru. Tento příkaz se zadává ve formátu `find /v /c "" file.txt`, kde `file.txt` je cesta k požadovanému souboru. Pokud cesta k souboru není zadaná, `type` začne číst ze standardního vstupu, dokud nepřečte znak `EOT`.

`sort`

`sort` na standardní výstup vypíše abecedně seřazené řádky zadaného souboru. Pokud cesta k souboru není zadaná, `sort` začne číst ze standardního vstupu, dokud nepřečte znak `EOT`.

`tasklist`

`tasklist` na standardní výstup vypíše tabulku procesů s informacemi o procesech, které mají v PCB záznam.

`shutdown`

`shutdown` ukončí všechny běžící procesy, tj. vč. všech spuštěných shellů.

`rgen`

`rgen` začne na standardní výstup vypisovat náhodné byty. Ne však byty s hodnotou 0x04, tj. znak `EOT`.

`freq`

`freq` začne číst ze standardního vstupu, dokud nepřečte znak `EOT`. Poté na standardní výstup vypíše frekvenční tabulku bytů pro všechny byty s frekvencí větší než 0.

`shell`

Příkaz `shell` spustí nový shell.

2.2.1 Spouštění uživatelských programů

Po zadání příkazu se pokusíme příslušný program spustit pomocí systémového volání `kiv_os_rtl::Clone_Process`. Jméno programu se nekontroluje, kontroluje se pouze návratová hodnota systémového volání. Pokud vše proběhlo v pořádku, v novém vlákně je spuštěn daný proces.

2.2.2 Orchestrace pipeline

Pokud je potřeba přesměrovat výstup procesu na vstup jiného pomocí rour, je potřeba spustit několik procesů po sobě. Procesy jsou spouštěny od posledního k prvnímu, tj. pokud vstup vypadá např. takto:

`> p1 | p2 | p3`,

potom prvním spuštěným procesem bude `p3` a posledním `p1`.

Procesy jsou spouštěny tímto způsobem z toho důvodu, že pokud uživatel zadá neplatný příkaz, potom je potřeba všechny (již spuštěné) procesy ukončit. Pokud by první proces v pipeline byl spuštěn a potom bylo nutné jej předčasně ukončit, jedinou možností by bylo poslat na jeho vstup znak `EOT`, protože standardní vstup shellu nechceme zavírat. Ale to s sebou nese komplikace. V případě spouštění od posledního k prvnímu taková situace nemůže nastat (pokud by první zadaný příkaz byl neplatný, nebude vůbec spuštěn).

Při předčasném ukončování procesů jsou zavřeny `Handles` vstupů a výstupů všech rour a otevřených souborů a nakonec jsou přečteny `ExitCodes` procesů.

Pokud jsou všechny zadané příkazy platné, jsou také všechny příslušné procesy spuštěny. Poté se začne čekat na skončení jednoho z procesů pomocí systémového volání `kiv_os_rtl::Wait_For`. Jakmile jeden z procesů skončí, přečte se jeho `ExitCode` a zavře se jeho vstup a výstup (pokud se jedná o roury, nebo soubory). Toto probíhá, dokud neskončí všechny takto spuštěné procesy.

3 Závěr

Zadáním semestrální práce byla simulace operačního systému. Zadání bylo z velké části splněno.

3.1 Rozdělení práce

Rozdělení práce mezi členy týmu vypadalo zhruba následovně (rozdělení není striktní, práce členů se v mnoha případech překrývala):

- Stanislav Král
  - implementace systémových volání, rour, PCB, VFS, práce v kernelu
- Eliška Mourycová
  - práce na programech v uživatelském prostoru (parsování, implementace), orchestrace pipelines
- Ondřej Drtina
  - práce na FS FAT s využitím přiloženého obrazu diskety, implementace VFS rozhraní

3.2 Zhodnocení dosažených výsledků

Přes veškerou snahu jsme bohužel nebyli schopni včas implementovat souborový systém FAT. Část jeho nutné funkcionality je implementovaná a funkční, ale nelze ho zatím integrovat do celého projektu. Přesto máme důvěru v robustnost a správnost implementace zbytku částí projektu.