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, vláken a otevřených souborů, souborový systém FAT s využitím přiloženého obrazu diskety, roury a přesměrování 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 4.1):

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
`exit`	Ukončí aktuální shell

Tabulka 1: Požadované příkazy shellu

2 Uživatelská příručka

Tato sekce popisuje ovládání simulátoru.

2.1 Spuštění simulátoru

Program spustíte dvojitým kliknutím na spustitelný soubor `boot.exe`, který se nachází v adresáři `compiled`. Zobrazí se terminálové okno, které je připraveno na přijímání příkazů.

Okno zobrazuje prompt (po spuštění ve formátu `C:\>`). V promptu se nastavuje informace o pracovním adresáři shellu.

2.2 Zadávání příkazů

Simulátor dokáže vykonat podporované příkazy (viz tabulku 1). Příkazy lze vykonat samostatně nebo je řadit do tzv. pipelines pomocí symbolu |, popř. přesměrovat obsah souboru na vstup prvního procesu pomocí < a výstup posledního procesu do souboru pomocí >. Např.:

```
C:\> p1 | p2 | p3 > out.txt < in.txt
```

Ukázka kódu 1: Příklad pipeline

Uživatel je informován o případné nemožnosti vykonání příkazu.

2.3 Ukončení simulace

Simulaci je možné ukončit příkazem `shutdown`. Pokud je spuštěný pouze jeden shell, příkaz `exit` také ukončí simulaci.

3 Kernel

Kernel je část operačního systému, která provádí inicializaci hardwaru, zajišťuje správu prostředků a umožňuje vytvářet programy či vlákna. Uživatelskému prostoru nabízí své služby pomocí tzv. systémových volání. V této semestrální práci lze kernel rozdělit do následujících částí:

- správa procesů/vláken
- správa otevřených souborů
- souborový systém FAT12

3.1 Správa procesů a vláken

Tuto část kernelu lze považovat za nejdůležitější, jelikož bez její přítomnosti by nebylo možné spouštět žádné programy. Stará se o vytváření procesů a jejich správu, kdy lze procesy synchronizovat a nastavovat jejich návratové hodnoty. Simulace vláken a procesů je realizována pomocí konstrukcí pro vytváření vláken ze standardní knihovny `thread`. Většina kódu této části se nachází v souboru `/kernel/process.cpp`.

Pro použití služeb této části kernelu z uživatelského prostoru slouží následující systémová volání ze skupiny `Process`.

- `Clone`,
- `Wait_For`,
- `Read_Exit_Code`,
- `Exit`,
- `Register_Signal_Handler`

3.1.1 Vytvoření nového vlákna

Nativní identifikátory vytvořených vláken jsou mapovány na typ `kiv_os::THandle`, který se dále používá v rámci kernelu jako interní identifikátor vláken.

Při zpracování požadavku na vytvoření nového vlákna se ze vstupních registrů načtou potřebné argumenty, a zavolá se funkce `run_in_a_thread`, která přebírá argument obsahující vstupní bod části kódu, jež se má spustit v novém vlákně. Tato funkce vytvoří nové `std::thread` vlákno, kterému jako vstupní funkci nastaví funkci `thread_entrypoint` přebírající skrz argumenty vstupní bod kódu ke spuštění. Uvnitř spuštěného vlákna se před vykonáváním

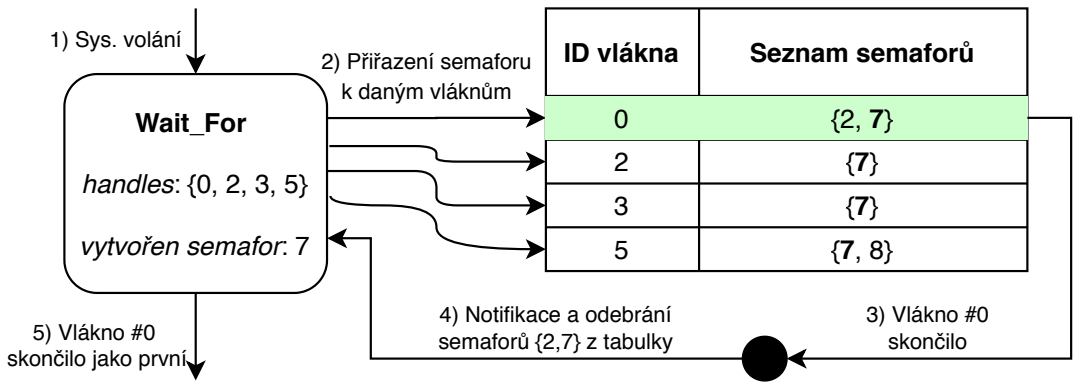
kódu dále čeká, dokud kernel nepřidá toto vlákno do tabulky všech vláken. Čekání je realizováno pomocí semaforu, a teprve po jeho notifikaci se začne vykonávat požadovaný kód.

Při vytváření nového vlákna se navíc dohledává, jaké vlákno či proces nové vlákno vytváří. Tato informace se později využívá např. při změně pracovního adresáře procesu.

3.1.2 Synchronizace vláken

Jádro umožňuje synchronizaci vytvořených vláken pomocí systémového volání `Wait_For`. Obsluha tohoto volání je realizována pomocí funkce `wait_for`, která přebírá pole obsahující identifikátory vláken, na která se má čekat. Synchronizace je realizována pomocí semaforů, kdy ke každému běžícímu vláknu je veden seznam semaforů, které se mají při skončení vlákna notifikovat. Jeden semafor může tedy být přiřazen k více než jednomu vláknu.

Na začátku obsluhy je k daným vláknům přiřazen nově vytvořený semafor, a zahájí se čekání na notifikaci tohoto semaforu. V moment, kdy je daný semafor notifikován, je vlákno čekající na semafor probuzeno, a tento semafor je odebrán ze všech ostatních seznamů, kde se vyskytuje. Spolu s notifikací semaforu je čekajícímu vláknu předána i informace o tom, jaké vlákno semafor probudilo.



Obrázek 1: Zjednodušený diagram obsluhy sys. volání `Wait_For`

3.1.3 Vytváření procesů

Vytváření procesů je principiálně stejné jako vytváření vláken a používá se stejných funkcí jako při vytváření nového vlákna. Hlavním rozdílem je to, že

až v jádře se převádí název programu na vstupní bod programu, který se předává funkci `run_in_a_thread`. Před zahájením vykonávání kódu programu je po vytvoření nového vlákna dle jeho identifikátoru přidán nový záznam do tabulky všech procesů. Dále také nový proces dědí pracovní adresář od procesu, kterým byl vytvořen.

Tabulka procesů, která je v kódu implementována třídou `Process_Control_Block`, obsahuje následující sloupce:

- `kiv_os::THandle handle` - identifikátor (**PID**[1]) procesu,
- `kiv_os::THandle std_in` - identifikátor standardního vstupu procesu,
- `kiv_os::THandle std_out` - identifikátor standardního výstupu procesu,
- `char *program_name` - název programu, který proces vykonává,
- `std::filesystem::path working_directory` - aktuální pracovní adresář procesu,
- `kiv_os::NOS_Error exit_code` - návratová hodnota procesu programu, který proces vykonává,
- `Process_Status status` - stav procesu, který může nabývat hodnot `Ready`, `Running`, `Zombie`

Předtím, než proces začne vykonávat kód programu, setrvává ve stavu `Process_Status::Ready`. Během vykonávání programu setrvává ve stavu `Process_Status::Running`. Pokud proces již dokončil vykonávání kódu programu, potom setrvává ve stavu `Process_Status::Zombie`, dokud si jiný proces nepřečte jeho návratovou hodnotu.

3.1.4 Nastavení návratové hodnoty procesu

Všem procesům je při jejich vytvoření nastavena výchozí návratová hodnota `kiv_os::NOS_Error::Success`. V případě, že nějaký program chce tuto hodnotu nastavit ručně, tak má možnost použít systémové volání `Exit`, a v registrech nastavit požadovanou návratovou hodnotu.

[1] process identifier

3.1.5 Přečtení návratové hodnoty procesu

Přečtení návratové hodnoty procesu z uživatelského prostoru je možné pomocí systémového volání `Read_Exit_Code`, kdy je před voláním nutné nastavit do registrů identifikátor procesu, jehož návratovou hodnotu chceme přečíst.

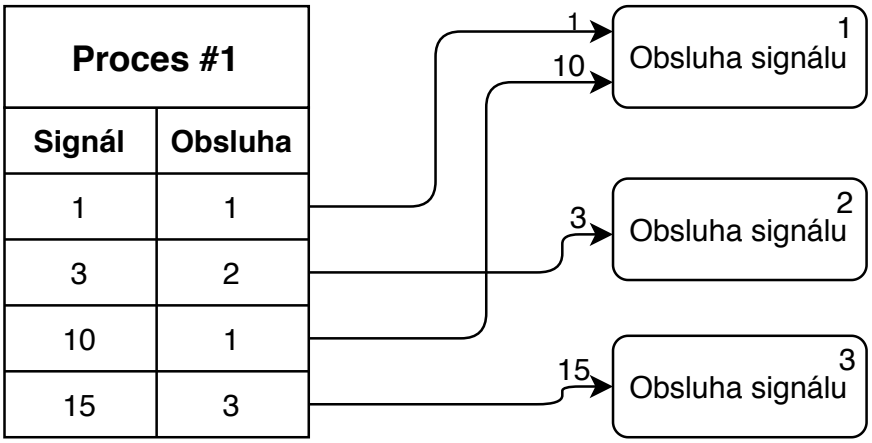
Toto volání je implementováno tak, že se nejdřív zkontroluje, jestli daný proces již skončil a setrvává ve stavu `Process_Status::Zombie`. Pokud ano, tak se jednoduše z tabulky získá jeho návratová hodnota. V případě, že proces ještě neskončil, tak v tomto stavu nesetrvává, a tudíž nelze v tuto chvíli přečíst jeho návratovou hodnotu, protože by se ještě mohla změnit. Obsluha systémového volání `Read_Exit_Code` tedy zavolá funkci `wait_for` a počká, dokud daný proces neskončí. Poté přečte a vrátí jeho návratovou hodnotu.

Na konci tohoto volání po úspěšném přečtení návratové hodnoty je z tabulky procesů daný proces, jehož návratovou hodnotu jsme přečetli, odebrán.

3.1.6 Nastavení obsluhy systémového signálu aktuálního procesu

Aby každý proces mohl nastavit vlastní obsluhu libovolného systémového signálu, jádro poskytuje možnost volat systémové volání
`Register_Signal_Handler`.

Kontrakt definovaný v `/api/api.h` umožňuje použít jednu obsluhu víckrát a nastavit ji pro více signálů. Implementace tohoto kontraktu je realizována tak, že každý proces má vlastní tabulku, kde klíčem je signál a hodnotou je adresa obsluhy daného signálu.



Obrázek 2: Vizualizace tabulky obsluh systémových signálů jednoho procesu

Při rozeslání signálu jádrem se dle tabulky obsluh každého procesu dohledává, která obsluha se má zavolat. Pokud některý z procesů nemá nastavenou

vlastní obsluhu daného signálu, zavolá se výchozí obsluha signálu definovaná kernelem.

3.2 Implementace rour

Aby bylo možné přesměrovávat výstup jednoho procesu do vstupu jiného, je třeba, aby jádro umožňovalo vytvářet roury, které mají vstup, výstup a paměťový buffer.

Implementace rour představuje rozšíření řešení problému *producent a konzument*. Třída, jejíž instance představují samotné roury, je definována v souborech `kernel/pipe.h` a `kernel/pipe.cpp`. V konstruktoru přijímá velikost bufferu, podle které dále inicializuje dva semafory. Volajícímu poskytuje následující metody:

- `std::vector<char> Pipe::Read(size_t read, bool &empty)` - tato metoda, která představuje konzumenta a slouží ke čtení požadovaného množství dat z bufferu, přijímá v druhém parametru referenci na booleanskou hodnotu, kterou nastaví na `true`, pokud byl přečten celý dostupný obsah bufferu a vstup roury byl zavřen. V takový moment již není co číst a nelze očekávat, že do bufferu bude cokoliv zapsáno, protože vstup byl již zavřen.
- `size_t Pipe::Write(std::vector<char> data)` - tato metoda představující producenta slouží k zapisování předaných dat do bufferu a vrací počet zapsaných bytů. Pokud byl v průběhu zápisu zavřen výstup nebo vstup roury, zapisování je přerušeno.
- `void Pipe::Close_Out()` - slouží k zavření výstupu, kdy po zavření notifikuje semafory pro zápis i čtení, aby bylo možné oznámit vláknům čekajícím na zápis nebo čtení událost zavření výstupu.
- `void Pipe::Close_In()` - slouží k zavření vstupu, kdy nejdřív zapíše znak `EOT` a až poté vstup zavře.

Pro vytvoření rour z uživatelského prostředí slouží systémové volání `Create_Pipe`, které vytvoří nový objekt roury, dvakrát jej vloží do tabulky souborů (jedno pro zápis a jednou pro čtení) a vygenerované identifikátory (indexy do souoborové tabulky) zapíše do registrů.

3.3 Tabulka souborů

K uložení informací o aktuálně otevřených souborech slouží tzv. tabulka souborů. Taková tabulka v sobě obsahuje seznam všech aktuálně otevřených

souborů, kdy každý záznam v tabulce představuje jedno otevření souboru a může obsahovat informaci o způsobu, jakým byl otevřen (např. atributy `kiv_os::NFile_Attribtues`). Tento záznam také obsahuje odkaz na instanci implementace abstraktní třídy `Generic_File`, která definuje, jak se z daného souboru čte. V této semestrální práci jsou implementovány následující typy souborů:

- soubor souborového systému – implementován ve třídě `Filesystem_File` a v konstruktoru přebírá odkaz na rozhraní souborového systému, kterým může být souborový systém FAT nebo `procfs`, který slouží ke čtení z tabulky procesů.
- soubor vstupu roury – implementován ve třídě `Pipe_In_File` a v konstruktoru přebírá odkaz na instanci roury. Tento soubor podporuje pouze zápis, a zapisuje do roury.
- soubor výstupu roury – implementován ve třídě `Pipe_Out_File` a v konstruktoru přebírá odkaz na instanci roury. Tento soubor podporuje pouze čtení, a čte z roury.
- soubor klávesnice – implementován ve třídě `Keyboard_File` a umožňuje číst z klávesnice, potažmo z konzole. Umožňuje i zápis, který zapisuje do virtuálního bufferu klávesnice.
- soubor textového výstupu – implementován ve třídě `Tty_File` a umožňuje zapisovat do konzole.

Tato tabulka je implementovaná jako třída obalující `std::map<kiv_os::THandle, std::unique_ptr<Generic_File»`. Otevřením nového souboru vznikne v této tabulce nový záznam, a při zavření je z tabulky odebrán.

Identifikátor	**Obecný souborový objekt**
1	`Keyboard_File`
2	`Tty_File`
3	`Pipe_In_File`
4	`Pipe_Out_File`
5	`Fs_File`

Tabulka 2: Ukázka tabulky obecných souborových objektů

Díky této implementaci je tedy zápis a čtení do souborů jednoduchý, protože logika implementace je volajícímu schovaná – stačí pouze znát identifikátor požadovaného souboru.

3.3.1 Operace nad soubory

Nad soubory lze provádět různé operace pomocí následujících systémových volání:

- `Read_File` – vyhledá v tabulce souborů daný soubor a provede čtení,
- `Write_File` – vyhledá v tabulce souborů daný soubor a provede zápis,
- `Close_File` – vyhledá v tabulce souborů daný soubor, zavře ho a odstraní ho ze souborové tabulky,
- `Seek` – vyhledá v tabulce souborů daný soubor, a dle zadaných parametrů provede jeho zvětšení nebo změnu aktuální pozice kurzoru,
- `Get_File_Attribute` – vyhledá v tabulce souborů daný soubor, a vrátí jeho atributy,
- `Seek_File_Attribute` – vyhledá v tabulce souborů daný soubor, a změní jeho atributy,
- `Delete_File` – vyhledá dle specifikované cesty v připojených souborových systémech soubor, a pokud existuje, tak se ho pokusí smazat.

3.4 Obecné rozhraní souborového systému

Každý souborový systém, který by má být podporován jádrem, musí implementovat rozhraní virtuálního souborového systému. Toto rozhraní je definováno v abstraktní třídě `VFS`. Definuje hlavičky metod pro následující operace:

- vytvoření nové složky
- odstranění složky
- přečtení obsahu složky
- otevření souboru (buď již existujícího nebo nově vytvořeného)
- zápis
- čtení
- odstranění souboru
- ověření existence souboru ve specifikované složce

- nastavení a získání atributů souboru

Tato třída implementuje metodu `generate_dir_vector`, která v parametrech přebírá seznam položek složky (seznam struktur `kiv_os::TDir_Entry`) a převede je na seznam znaků (`char`), aby bylo možné tento seznam přečíst z uživatelského prostoru.

Tento způsob definice rozhraní souborového systému umožňuje flexibilní přidávání dalších podporovaných systémů, kdy stačí pouze implementovat dané rozhraní a definovat, na jaké cestě se nachází kořen souborového systému.

3.4.1 Ověření existence souboru

Z uživatelského prostoru může přijít požadavek na otevření souboru, jehož adresa může být specifikována buď jako relativní nebo absolutní. Pro jednodušší práci s adresami se používá typ `std::filesystem::path` ze standardní knihovny. Pokud byla specifikována relativně, tak se vždy na začátek této adresy ještě přidá aktuální pracovní adresář, a začne ověřování existence jednotlivých komponent cesty.
Ověření existence cesty `A` používá následující algoritmus:

1. načti komponentu adresy a přidej jí do `P`
2. zjisti, jaký souborový systém `S` se na `P` používá,
3. ověř existenci souboru na cestě `P` pomocí `S`
4. pokud na `P` v `S` soubor neexistuje – **cesta je neplatná**
5. pokud existuje a cesta obsahuje další komponentu, běž na **1.**
6. pokud existuje a cesta již neobsahuje další komponentu, `P` **ukazuje na platný soubor** v `S`

Při změně aktuálně používaného souborového systému se do ADT struktury zásobník přidá na vrchol dosud používaný souborový systém. V každém souborovém systému se počítá počet zanoření. Zpracování komponenty `..` snižuje počet zanoření v stromové struktuře složek. Pokud počet zanoření dosáhne nulové hodnoty, tak se ze zásobníku vyjme poslední používaný souborový systém. Využití zásobníku snižuje počet vyhledávání kořenů souborových systémů.

3.5 Pracovní adresář

Pomocí systémových volání `Set_Working_Dir` a `Get_Working_Dir` se mění a získává pracovní adresář aktuálního procesu (tj. toho, který inicializoval systémové volání). Při změně adresáře se pomocí algoritmu ukázaném v podkapitole 3.5 kontroluje existence adresáře nacházajícího se na zadané cestě. Dále se také kontroluje, zdali cesta opravdu ukazuje na adresář, a ne například na obyčejný soubor.

3.6 Souborový systém `procfs`

Aby bylo možné nechat z uživatelského prostoru vypsat obsah tabulky procesů a zobrazit tak seznam aktuálně běžících procesů, je třeba, aby v kernelu byl implementován vlastní souborový systém, který bude číst z této tabulky, a pomocí kombinace virtuálních souborů a adresáře tak zpřístupňovat seznam procesů.

V této semestrální práci se na adrese `C:\procfs` nachází adresář, jehož položky představují položky tabulky procesů a jsou pojmenovány identifikátory procesů. Při čtení položek z tohoto adresáře jsou položky tabulky procesů převedeny na strukturu `PCB_Entry` a uloženy v paměťovém bufferu. Dle parametrů předaných při čtení je volajícímu vrácen kus paměťového bufferu.

Jelikož v souboru `api.h` není definována struktura položky tabulky procesů, tak by uživatelský prostor nevěděl, jak obsah souborů v adresáři `procfs` interpretovat. Z tohoto důvodu byla po dohodě s vyučujícím do kernelu i uživatelského prostoru přidána struktura `PCB_Entry`, kterou kernel vrací při čtení z adresáře `procfs`, a tak uživatelský prostor ví, jak obsah souborů interpretovat.

3.6.1 Alternativní řešení

Sdílení struktury mezi uživatelským prostorem a kernelem lze předejít tak, že by položky v adresáří `procfs` představovaly adresáře, které by se jmenovaly dle identifikátorů běžících procesů. V těchto adresářích by se dále nacházely soubory, kdy každý soubor by obsahoval jeden atribut daného procesu.

3.7 Implementace souborového systému `FAT12`

Program umožňuje vytváření souborů a složek, přičemž k jejichž uchování je využíván dodaný obraz diskety pracující se souborovým systémem `FAT12`. Funkce, jež implementují uvedený souborový systém, jsou definovány v souborech `kernel/fat_fs.h` a `kernel/fat_fs_utils.h`.

3.7.1 Definice v `kernel/fat_fs.h`

Obsahuje definici veškerých funkcí, které jsou volány vyššími vrstvami OS ve spojitosti s prací se souborovým systémem `FAT12`. Některé z funkcí, jež jsou v souboru uvedeny, ve svém těle volají funkce definované v souboru `kernel/fat_fs_utils.h`.

Třída `Fat_Fs` dědí od abstraktní třídy `VFS` a obsahuje funkce umožňující zápis, respektive čtení souboru. Rovněž jsou obsaženy funkce, které umožňují vyšším vrstvám OS vytváření a mazání složek. Obsažena je i funkce ověřující existenci souborů a složek.

3.7.2 Definice v `kernel/fat_fs_utils.h`

Obsahuje podpůrné funkce souborového systému `FAT12`, jež nejsou přímo volány vyššími vrstvami OS, ale jsou využívány v rámci třídy `Fat_Fs`. Jsou zde obsaženy funkce usnadňující práci se souborovým systémem. Příklady obsažených funkcí:

- `int allocate_new_cluster()` - alokace nového clusteru (souboru / složky)
- `std::vector<int> retrieve_sectors_nums_fs` - získání seznamu sektorů daného souboru či složky
- `bool check_file_name_validity()` - zjištění platnosti názvu souboru ve `FAT12`.

3.7.3 Problémy spojené s implementací

Při implementaci souborového systému byl pro nás problémem zejména fakt, že ve `FAT12` tabulce, jež obsahuje informace o stavu clusterů, je jedna hodnota reprezentována 12 bity. Při editaci hodnot v tabulce bylo tedy třeba načíst vždy dva bajty (resp. 16 bitů) a upravit jen odpovídající část dat.

4 Uživatelský prostor

Uživatelský prostor je část operačního systému, která pomocí systémových volání žádá jádro 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`.

4.1 Podrobný popis podporovaných příkazů

Zde je podrobnější popis podporovaných příkazů. Implementace funkcionality se 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 2: Ukázka chování programu echo

Pokud je argument příkazu `echo` řetězec `on` nebo `off`, `echo` se nevykoná jako externí program, pouze skryje prompt aktuálního shellu.

`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.

`cd` se nevykonává jako externí příkaz, pouze nastaví pracovní adresář aktuálního shellu.

`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.

`rd`

`rd` smaže adresář zadaný v argumentu.

`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 /c /v"" 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.

`tasklist` získá informace z adresáře `/procfs`. V tomto adresáři se nachází několik položek s názvy `/procfs/pid`, kde `pid` je PID procesu, který má záznam v PCB. Mezi jádrem a uživatelským prostorem se nachází sdílená

struktura `PCB_Entry`. Každý ze souborů `/procfs/pid` obsahuje jednu strukturu `PCB_Entry`, `tasklist` tedy z každého souboru přečte velikost `PCB_Entry` bajtů a výsledek přetypuje na `PCB_Entry`. Získané informace naformátuje do výsledné tabulky.

`shutdown`

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

`rgen`

`rgen` začne na standardní výstup vypisovat náhodná čísla v plovoucí čárce.

`rgen` používá dvě vlákna - jedno pro generování čísel a jejich výpis a druhé pro čtení vstupu, aby poznal, jestli byl na jeho vstup zapsán znak `EOT`, `ETX` nebo `EOF`.

`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.

`exit`

Příkaz `exit` ukončí aktuální shell.

4.2 Spouštění uživatelských programů

Při zadání vstupního řetězce uživatelem je řetězec rozdělen na jméno programu a argumenty (popř. jména, pokud vstup obsahuje symbol roury nebo přesměrování).

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.

Po skončení procesu je přečten jeho návratový kód pomocí systémového volání `kiv_os_rtl::Read_Exit_Code`.

4.2.1 Orchestrace pipeline

Pokud je potřeba přesměrovat výstup procesu na vstup jiného pomocí rour, je potřeba spustit několik procesů po sobě.

Nejdříve je vytvořeno dostatečné množství rour pomocí systémového volání `kiv_os_rtl::Create_Pipe`. `Handles` vstupů a výstupů jednotlivých rour jsou uloženy.

Procesy jsou spouštěny od posledního k prvnímu, tj. pokud vstup vypadá např. takto:

`C:\> 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 zapsat na jeho vstup znak `EOT`, protože standardní vstup shellu nechceme zavírat. To s sebou ale 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.

5 Závěr

Zadání semestrální práce bylo splněno. Byly navrženy a implementovány všechny jednotlivé části popsané v Zadání.

5.1 Rozdělení práce a bodů

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

- Stanislav Král (1/3 bodů)
  - implementace systémových volání, rour, PCB, VFS, práce v kernelu
- Eliška Mourycová (1/3 bodů)
  - práce na programech v uživatelském prostoru (parsování, implementace), orchestrace pipelines
- Ondřej Drtina (1/3 bodů)
  - práce na FS FAT s využitím přiloženého obrazu diskety, implementace VFS rozhraní