FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

KIV/OS

1. Simulace operačního systému

Patrik Patera - A17N0083P,
Luděk Kaňák - A17N0071P,
Tomáš Šimandl - A17N0088P

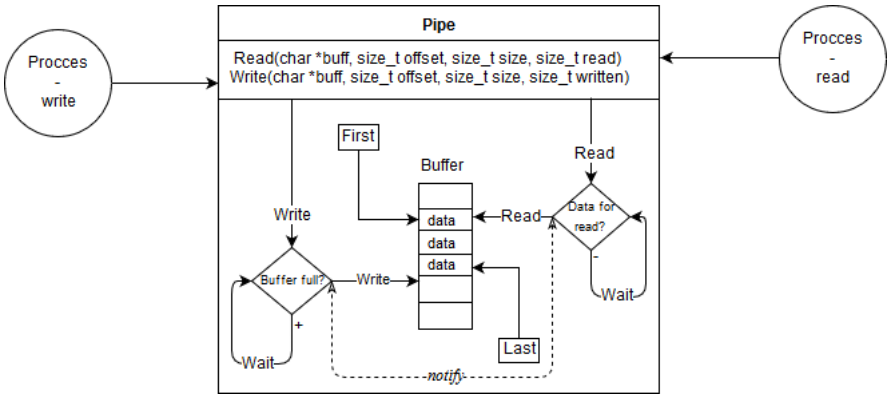
Email: pat.patera@gmail.com

27.11.2017

Obsah

1 Roura

Roura (pipe) slouží jako synchronizační prostředek mezi procesy a funguje na bázi producent–konzument. Součástí roury je kruhová vyrovnávací paměť (buffer), jejíž velikost je nastavena na hodnotu 4096 bytů a jedná se o pole, které je datového typu `char`. Díky této paměti je umožněno jednomu procesu do ní zapisovat a druhému procesu z ní číst, proto je také nutné vytvořit dva souborové deskriptory typu `PipeHandler` (jeden pro čtení a druhý pro zápis). Dvojice proměnných `first` a `last` určují pozici ve vyrovnávací paměti, kde zapsaná data začínají a končí. Při zápisu se mění pouze pozice `last`, která udává poslední pozici zapsaných dat v paměti. Naopak při čtení se mění pouze pozice `first`, která udává, kde data v paměti začínají. Pokud by vyrovnávací paměť byla celkově zaplněna a proces by se pokusil o zápis, pak je zablokován do té doby dokud ho proces, co z paměti čte, nevzbudí nebo neuzavře svůj deskriptor (přečetl data z paměti a tak uvolnil část paměti). Roura je uzavřena ve chvíli, kdy proces, který zapisoval do paměti, uzavře souborový deskriptor a to vygeneruje EOF, který je přečten druhým procesem k signalizaci ukončení souborového deskriptoru pro čtení. Výhodou roury je, že výstup jednoho procesu je takto přesměrován na vstup druhého procesu, jak je možné vidět na obrázku 1.



Obrázek 1: Detailnější znázornění roury.

2 Zpracování příkazů

O zpracovaní veškerých příkazů se stará třída `CommandsWrapper`, kterou si každý nově spuštění `shell` vytvoří. Třída `CommandsWrapper` má na starosti celý proces zpracovaní řetězce z `stdOut`. Pokud by během zpracovaní jednotlivých částí došlo k nekonzistenci s gramatikou `cmd`, pak je celý proces zpracovaní ukončen a je vypsána příslušná chyba na `stdErr`, není tak zbytečně voláno jádro operačního systému s nevalidními hodnotami (voláno pouze `Write_File` z `rtl` pro výpis chyby).

Celý proces je rozdělen do několika částí, které mají na starosti zpracovat různé sekvence vstupního řetězce.

- V první řadě je řetězec rozdělen podle znaku |, který reprezentuje jednotlivé roury. Celý podřetězec nacházející se před tímto znakem je dále poslán k zpracovaní. V případě, že se kolem znaku roury nenacházejí mezery, pak je vypsána chyba – špatná syntaxe. Pokud se ovšem tento znak nachází mezi znakem "a ", pak je to bráno jako, že je součástí řetězce. Řetězec musí být určen jak počátečním ", tak u ukončovacím znakem ".

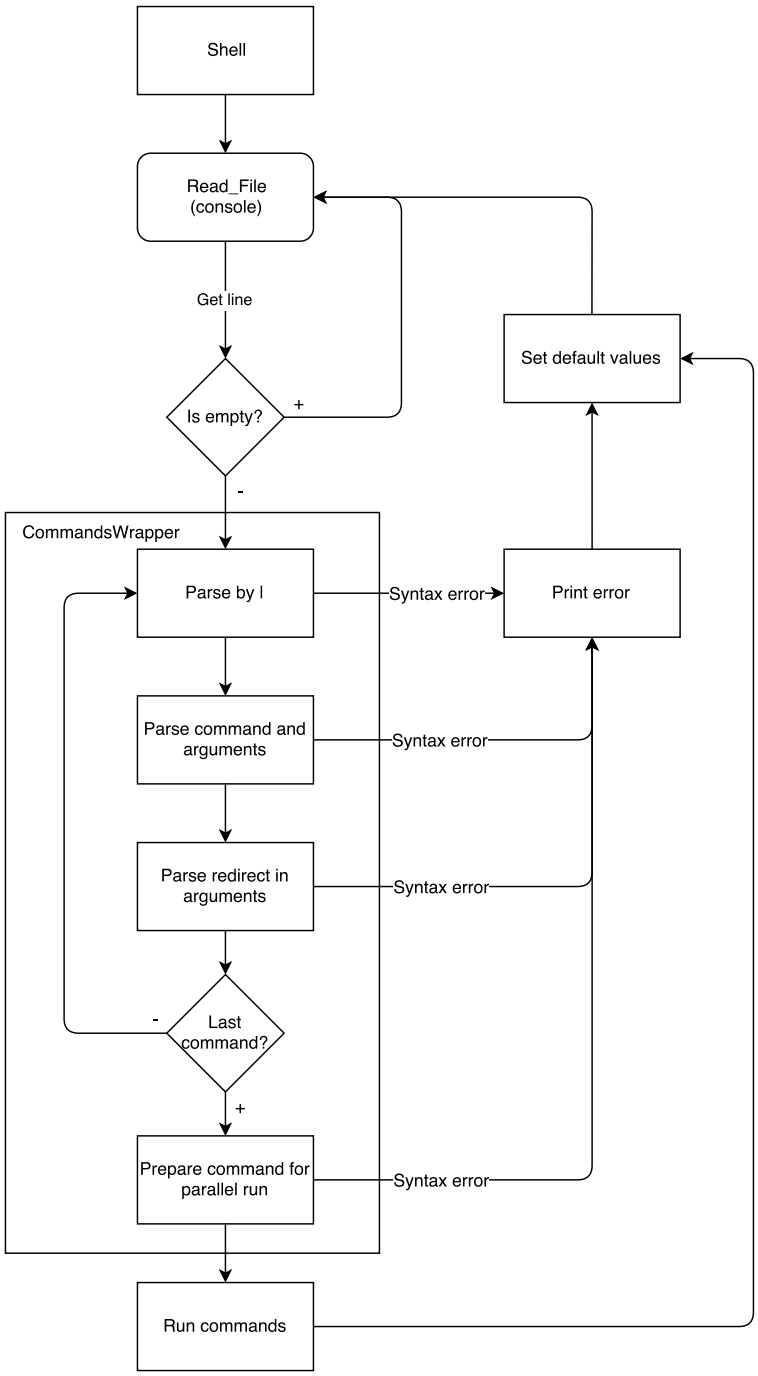
- V následující části je z podřetězce nalezen příkaz a zbytek podřetězce je brán jako vstupní argumenty daného programu. Tyto položky jsou uloženy do struktury `cmd_item_t` pro pozdější zpracovaní a vložen do seznamu celkových příkazů.

- Nyní je zkoumáno, jestli se v argumentech nacházejí nějaké přesměrování. Jsou uvažovány obě možnosti, a to ze souboru `<` či do souboru `>` nebo `>>`. V případě, že před a za těmito znaky není mezera, je vypsána chyba a celý proces ukončen – špatná syntaxe. Pokud je přesměrování zapsáno správně jsou informace opět uloženy do struktury `cmd_item_t`, která již byla založena v předchozí části. Pokud argument neobsahuje žádné přesměrování, pak se tato část nevykonává a je pokračováno dále ve zpracovaní.

- Pokud vstupní řetěz obsahuje další znak `|`, pak je celý tento proces zopakován do té doby, dokud není celý vstupní řetězec zpracován.

- Na konci se provede příprava všech příkazů uložených v seznamu. Jsou vytvořeny souborové deskriptory (čtení a zápis) pro každou rouru, případně pro přesměrování. Každý z těchto příkazu spustí paralelně svůj program v procesu, kterému jsou předány příslušně souborové deskriptory, aby bylo zajištěno přesměrování výstupu jednoho procesu na vstup druhého procesu. Po ukončení, ať už úspěšném či neúspěšném, je možné zpracovat další vstup z `stdIn`.

Celý tento proces popisuje Obrázek 2.



Obrázek 2: Diagram průběhu zpracování příkazu

3 File Allocation table (FAT)

Jako souborový systém jsme použili FAT postavený na semestrální práci z KIV/-ZOS. Původní práce byla rozšířena o zápis do již existujícího souboru, čtení souboru od dané pozice a nastavení velikosti souboru. Dalším rozdílem je realizace samotného disku. V původní práci byl disk reprezentován souborem uloženým v souborovém systému, nyní je disk reprezentován vstupním polem, které je při spuštění systému formátováno jako FAT disk. Pole je uloženo ve třídě `FatFS` a je do modulu `FAT` předáváno při volání jednotlivých funkcí.

4 Souborový systém (FS – File system)

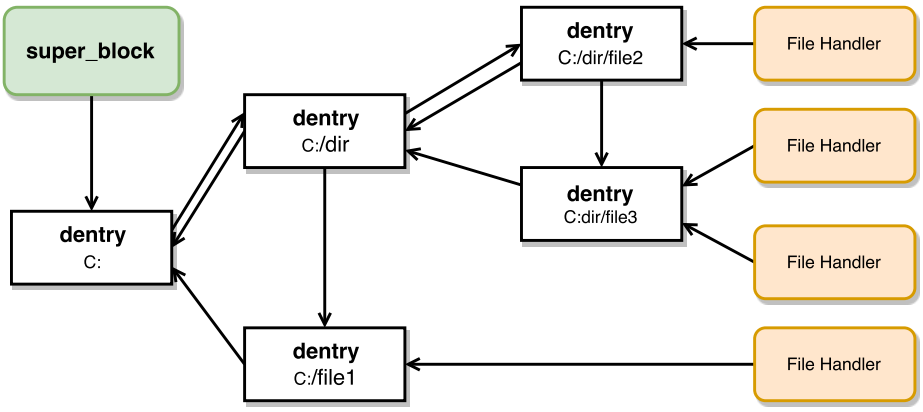
Nad všemi souborovými systémy je abstraktní třída `FS`. Třída obsahuje abstraktní metody, které jsou volány z `VFS` a z `FileHandleru`. Každý souborový systém musí tyto metody implementovat nebo spíše vytvořit mezivrstvu která tyto metody obsahuje a používá metody daného souborového systému. `FS` obsahuje strukturu `super_block` obsahující základní informace o souborovém systému jako například `id, root adresář, velikost bloku, atd.`. Dále v této kapitole budou pod pojmem soubor myšleny i adresáře. Každý otevřený soubor je definován strukturou `dentry`, která obsahuje veškeré informace, které jsou potřeba pro práci se souborem. Jednotlivé záznamy `dentry` jsou řazeny do struktury vyobrazené na obrázku 3. Zároveň s otevřeným souborem jsou uloženy ve struktuře veškeré adresáře v cestě od kořene k danému souboru. Výhodou je nemožnost smazání rodičovského adresáře daného souboru. Ke každému otevřenému fyzickému souboru v souborovém systému náleží jeden záznam `dentry` a zároveň na jeden záznam (`dentry`) může odkazovat více `File_Handlerů`(viz kapitola 8.1). Při uzavření souboru, není-li otevřen v jiném `File_Handleru`, je záznam `dentry` ze struktury vymazán a spolu s ním i nevyužívané záznamy v cestě k souboru. Při uzavření souborového systému se projde celá struktura a uzavřou se veškeré načtené záznamy.

4.1 Fat souborový systém (FatFS)

Třída dědí od `FS` a vytváří tak mezivrstvu mezi `VFS` a modulem `FAT`. Jelikož `FAT` slouží spíše jako knihovna a je bezstavová, obsahuje také strukturu obsahující informace nutné pro práci s `FAT`. Struktura obsahuje pole formátované jako FAT disk, boot_record, fat tabulky a další konstanty načtené při inicializaci `FAT`. Hlavním úkolem této třídy je převádět data z modulu `FAT` do čitelných dat pro `VFS`.

5 Virtual file system (VFS)

Virtual file system je postaven nad všemi souborovými systémy. Každý FS musí být registrován u VFS. Ten je uložen do mapy všech registrovaných FS v systému. Mapa používá jako klíč název FS (například C:). Důležité je aby název



Obrázek 3: Zobrazení struktury načtených souborů ve `FS`

disku končil dvojtečkou. Název musí být unikátní a název "0:"je zabrán pro `Process File System` (viz kapitola 10.1). Při ukončení systému jsou všechny registrované souborové systémy odstraněny. VFS je volán z IO a slouží pro otevírání, vytváření a mazání souborů. Funkce pro zápis nebo čtení ze souboru jsou zprostředkovány pomocí třídy `FileHandler`(viz kapitola 8.1). Všechny funkce obsahují jako vstupní argument absolutní cestu, kdy je nejprve podle cesty nalezen FS a až poté je předán požadavek konkrétnímu FS.

6 Služby IO

IO je vstupní vrstvou pro všechny io požadavky z uživatelského režimu. Nejprve je z registru zjištěno číslo volané služby. Každá služba poté volá metody z `VFS`, `Handleru` nebo `ProcessManageru`. Všechny služby při jakékoliv chybě nastaví carry bit a do registrů uloží kód chyby.

Služby:

Create_File - z registrů je načtena cesta k souboru a další atributy. Neobsahuje-li cesta název souborového systému je k cestě připojen aktuální pracovní adresář, který je získán z `ProcessManageru`. Na základě atributů je buďto otevřen nebo vytvořen soubor či adresář. Nebyla-li vrácena chyba, je vytvořený handler předán do `ProcessManageru` a číslo handleru je uloženo do registrů a tím vráceno do uživatelského režimu.

Write_File - nejprve je z registrů získáno číslo handleru. Číslo je předáno do `ProcessManageru`, který vrátí příslušný handler. Jelikož je `Handler` abstraktní třída, je pouze zavolána metoda `Write` a podle instance se zavolá `FileHandler`, `PipeHandler` nebo `Console`. Výsledek je předán do registrů.

Read_File - služba je velice podobná službě `Write_File` jen s tím rozdílem, že místo volání metody `Write` je volána metoda `Read`. Výsledek je opět předán do registrů.

Delete_File - z registrů je načtena cesta k souboru. Neobsahuje-li cesta název, souborového systému je k cestě připojen aktuální pracovní adresář, který je získán z `ProcessManageru`. Jelikož api není připravené na možnost stejného názvu pro soubor i pro adresář v jednom adresáři, je nejprve smazán soubor voláním VFS metody pro smazání souboru. Nebyl-li soubor nalezen, zavolá se metoda VFS pro smazání adresáře. Nebyl-li ani tak soubor odstraněn, je do registrů vložen kód chyby a je nastaven carry bit.

Set_File_Position - nejprve je z registrů získáno číslo handleru a další atributy. Číslo je předáno do `ProcessManageru`, který vrátí příslušný handler. Jelikož je pozice v souboru různá pro každý handler, nevolá se VFS, ale je volána metoda handleru `fseek`.

Get_File_Position - stejně jako služba `Set_File_Position` získá handler nad kterým je tentokrát zavolána metoda `ftell`. Výsledek je uložen do registrů.

Close_Handle - služba pouze získá z registrů číslo handleru a předá jej do `ProcessManageru`. Ten jej dealokuje a odstraní. Každý handler má definovaný svůj destruktor, který se postará o dealokaci veškeré příslušné alokované paměti.

Get_Current_Directory - aktuální pracovní adresář je uložen v PCB tabulce procesu. Je tedy volán `ProcessManager`, který vrátí pracovní adresář. Nevejde-li se řetězec určující pracovní adresář do připraveného vstupního pole znaků, je vrácena chyba. V opačném případě je řetězec zkopírován do pole a velikost řetězce je uložena do registrů.

Set_Current_Directory - aby nebyl uložen nevalidní pracovní adresář je nejprve volána metoda na otevření souboru z VFS. Je-li adresář úspěšně nalezen je řetězec udávající pracovní adresář předán do `ProcessManageru`.

Create_Pipe - je vytvořena nová roura. Poté jsou vytvořeny dva handlery. Jeden pro zápis a druhý pro čtení z roury. Handlery jsou předány do `ProcessManageru` a čísla handlerů jsou uložena do registrů pro návrat do uživatelského režimu.

7 Inicializace VFS a FS

Nejprve je potřeba vytvořit pole dostatečné velikosti, které bude představovat disk. Doporučená minimální velikost je 4096 bytů. V naší práci jsme však zvolili velikost 65536 bytů. Pole je poté formátováno statickou metodou třídy `FatFS`

`init_fat_disk`. Vstupními parametry jsou vytvořené pole, velikost pole a velikost jednoho bloku. Velikost bloku ovlivní maximální počet souborů v adresáři jelikož adresář zabírá přesně jeden blok.

Inicializace FAT souborového systému

Do třídy `FatFS` je předáno formátované pole, velikost pole a název svazku. Konstruktor vytvoří strukturu obsahující informace o fat disku. Do struktury je předáno vstupní pole a celá struktura je předána do funkce `fat_init` v modulu `fat`. Tam je z pole načten `boot_record`, obě fat tabulky a jsou spočítané často využívané konstanty. Nakonec je vše uloženo do vstupní struktury. Ve třídě `FatFS` je z načtených dat inicializován `super_block` a kořenový adresář. Kořenový adresář je vložen do `super_blocku` a ten je uložen ve `FatFS` pro pozdější užití.

Inicializace VFS a registrace FS

Inicializace VFS pouze inicializuje mapu souborových systémů. Poté je volána metoda třídy `VFS register_fs` do které je předán název registrovaného souborového systému a instance souborového systému. Při registraci je nejprve prohledána mapa souborových systémů zda již neobsahuje souborový systém s daným názvem. Poté je nový systém přidán do mapy.

8 Handlery

Třída `Handler` představuje společné rozhraní pro všechny handly týkající se zápisu a nebo čtení. To umožní pracovat se těmito handly stejným způsobem. Konkrátně se jedná o `File_Handler`, `Pipe_Handler` a `Console`. Třídy, které toto rozhraní dědí musí překrýt metody pro zápis, čtení a změnu pozice.

8.1 File Handler

Třída `File_Handler` dědí od abstraktní třídy `Handler`. Třída navíc obsahuje strukturu `dentry` představující záznam o otevřeném souboru. Metody `read` a `write` získají ze struktury `dentry` konkrétní souborový systém nad kterým jsou volány příslušné metody. Metoda `fseek` nastaví pozici souboru podle vstupních parametrů. Je-li nová pozice větší než velikost souboru nebo naopak menší než nula, je vrácena chyba. Je-li zároveň požadována změna velikosti souboru na aktuální pozici, je získán z `dentry` příslušný souborový systém a je zavolána příslušná metoda souborového systému.

8.2 Pipe handler

Třída `Pipe_Handler` dědí od abstraktní třídy `Handler`. Třída navíc obsahuje ukazatel na příslušnou rouru. Metody pro zápis a čtení jsou překryty, aby byl za jištěn zápis a čtení do vyrovnávací paměti roury. Po uzavření posledního pipe

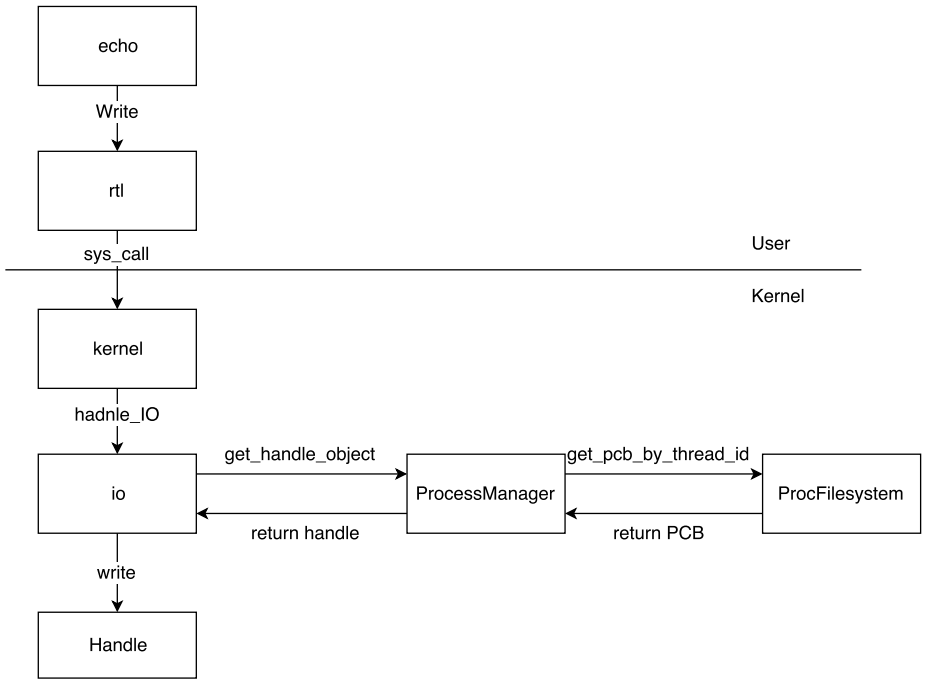
handleru (jsou vždy vytvořeny dva – čtení a zápis) je také zajištěno odstranění a uvolnění paměti po příslušné rouře.

8.3 Console handler

řída `Console` dědí od abstraktní třídy `Handler`. `Console` se může otevřít v režimu pro čtení, výpis a výpis chyby. Při čtení se kotroluje, zda nebyla zadána zkratka `Ctrl + z` a přečtené znaky se ukládají do **bufferu**. Pokud byla zkratka přečtena, ale mimo ní obsahuje načtený řetězec další znaky, tak se zkratka ignoruje snížením počtu přečtených znaků o 3. Pokud byla přečtena pouze zkratka, nastaví se počet přečtených znaků na 0, což znamená `EOF`. Při zápisu se zapíše daný počet znaků ze vstupního `bufferu`.

9 Runtime library

Runtime library (`rtl`) obsahuje funkce, ve kterých se naplní příslušné registry daty podle požadované služby a následně se provede systémové volání viz obrázek 4. Jádro obslouží požadavek a v registrech vrátí výsledek. Poté se v `rtl` přečte výsledek z registrů. V případě, že nastala chyba v jádře během zpracování výsledků, se uloží v `rtl` chybový kód, jenž se může po zavolání funkce `print_error` poslat na chyboví výstup. Jednotlivé funkce vrací informaci o výsledku v podobně `true/false`.



Obrázek 4: Ukázka volání funkce `write` v programu `echo`

10 Procesy

10.1 Process Filesystem

Informace o procesech a vláknech jsou uloženy ve dvou tabulkách. Tabulka pro procesy uchovává položky `PCB` a v případě vláken se jedná o tabulku obsahující položky `TCB`. Položka `PCB` obsahuje pro každý proces informace jako je: id procesu, id rodiče ve kterém byl proces spuštěn, název spuštěného programu, pracovní adresář a list s otevřenými soubory (`handly`). Položka pro vlákno se skládá z id vlákna, vlákna (`std::thread`) a odkazu na `PCB` procesu, ve kterém bylo vlákno spuštěno.

Tyto dvě tabulky jsou uchovány ve třídě `ProcFilesystem`, která je registrována u VFS jako souborový systém `ProcFilesystem` s názvem `0:`. Z rozhraní `FS` využívá pouze metodu pro vytvoření souboru, která vrátí platný `FileHandler` jen v případě, že se žádá o adresář `procfs`, a metodu pro čtení, která do bufferu zapíše informace o procesech.

Třída `ProcFilesystem` dále obsahuje metody pro přidání procesů a vláken do tabulek, mazání vláken, kde při smazání posledního vlákna odkazujícího na proces smaže z tabulky procesů informace o tomto procesu, a metody pro získání `PCB` podle `thread_id` a `TCB` podle čísla handlu (`tid`).

10.2 Process Manager

Třída ProcessManager obsluhuje všechny události ohledně procesů a vláken. K uchování a práci s procesy/vlákny volá metody z již popsané třídy ProcFilesystem viz 3.

10.2.1 Inicializace

Při spuštění systému po inicializaci jádra se jako první proces spustí `INIT`. Proces má `pid` nastavený na 1 a rodičovský pid (`ppid`) na 0. Dále se zde nastavuje pracovní adresář `C:` a inicializuje se vstup a výstup týkající se konzole. Po nastavení se proces přidá do tabulky procesů ve třídě `ProcFilesystem`. Poté se spustí ve vlákně tohoto procesu funkce, která spustí uživatelský program `shell`, ve kterém se následně můžou spouštět pomocí příkazů další programy. Proces `INIT` končí až po skončení prvního vytvořeného shellu.

10.2.2 Vytvoření nového procesu

Z registrů se zjistí název programu a struktura `TProcess_Startup_Info`, která obsahuje vstupní parametry pro program a identifikátory handlů pro vstup, výstup a chybu. Dále se zjistí, v jakém procesu se má nový proces spustit. Pro nalezení kontextu procesu se zjistí id `(thread_id)` zrovna běžícího vlákna a následně se najde v `ProcFilesystemu` v tabulce vláken odpovídající položka `TCB` s vláknem označeným stejným `thread_id`. Nalezené TCB ukazuje na položku PCB představující aktuální proces. Tento proces bude nastaven jako rodičovský

proces pro nově vytvářený proces. Poté se vytvářenému procesu nastaví handlery z rodičovského procesu podle identifikátorů uložených v `TProcess_Startup_Info`.

Po nastavení všech informací o procesu do PCB, se připraví položka `TCB` s odkazem na nový proces a ve vlákně se spustí funkce `run_process`, ve které se spouští uživatelský program s registry obsahující argumenty po jehož dokončení se uzavřou všechny handly. Toto vytvořené vlákno se uloží do `TCB`.

Pokud vytvoření procesu proběhlo bez problémů, tak se do registrů uloží id vytvořeného `TCB`, které představuje handle vlákna.V případě výskytu chyby se do registrů uloží informace o chybě.

10.2.3 Vytvoření nového vlákna

Z registrů se přečte vstupní bod programu vlákna a jeho vstupní argument data. Podobně jako u procesů se zjistí v jakém procesu se má vlákno spustit a vytvoří se položka TCB s odkazem na tento proces. Dále se ve vlákně spustí funkce `run_thread`, která zavolá funkci získanou z registrů. Do registrů se stejně jako u procesů uloží handle na vlákno a nebo kód chyby, pokud nastane.

10.2.4 Čekání na handly

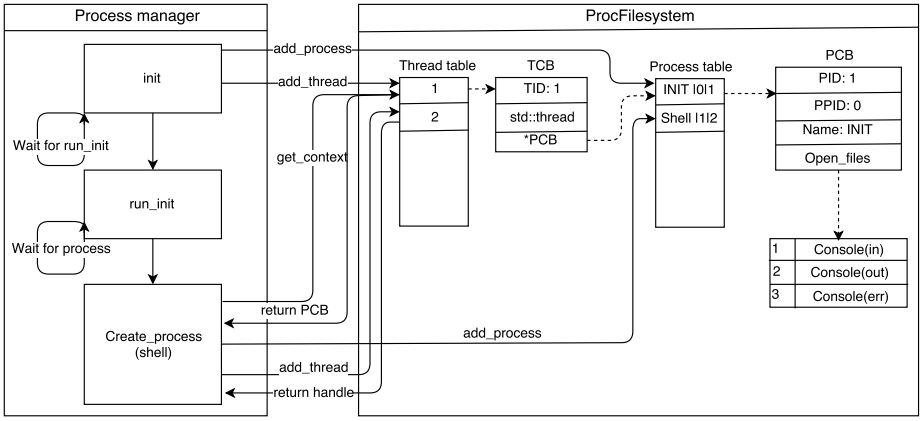
Z tabulky vláken se pro každý handle v poli najde příslušná položka `TCB`, nad uchovaným vláknem se zavolá `join` a čeká se dokud vlákno neskončí. Po skončení všech vláken s handly obsaženými v poli se vrátí handle na poslední proces.

10.2.5 Přidání a uzavření otevřených souborů

V metodách pro přidání (`add_handle`) a uzavírání (`close_handle`) souborů se v první řadě zjistí PCB procesu, ze kterého byly volány, stejným způsobem jako u vytváření procesu.

Při přidávání se otevřený soubor dá na první volné místo v poli s otevřenými soubory v PCB. Pokud není volné žádné místo, soubor se přidá na konec. Vrací se handle udávající index v poli nově přidaného souboru.

Otevřený soubor se uzavírá podle čísla handlu. Číslo handlu udává index v poli s otevřenými soubory procesu. Pokud se na pozici nachází neuzavřený soubor, tak se smaže a funkce vrátí hodnotu `true`, jinak vrátí `false`, což znamená neplatný handle.



Obrázek 5: Znázornění vytváření procesů

11 Příkazy

Vyskytne-li se v průběhu vykonávání jakéhokoliv příkazu chyba, je okamžitě vypsána do `stdError` a vykonávání příkazu je ukončeno.

11.1 Příkaz cd

Příkaz `cd` slouží primárně pro změnu pracovního adresáře. Syntaxe příkazu je: `cd [/d] [drive:][path]`. Při spuštění příkazu je nejprve kontrolováno, zda jsou vstupní parametry prázdné. Pokud ano, vypíše se pracovní adresář a příkaz se ukončí. Dále se hledá, zda je přítomen parametr `/d`. Zbytek argumentů je brán jako cesta a program pokračuje jednou z následujících větví:

Cesta začíná zpětným lomítkem - je voláno jádro pro získání pracovního adresáře z kterého je získán aktuální disk ke kterému je připojena vstupní cesta.

Cesta neobsahuje dvojtečku - je voláno jádro pro získání pracovního adresáře, ke kterému je připojena cesta z argumentů.

Ani jedna z výše uvedených - nebyl-li nalezen parametr `/d` provede se změna adresáře pouze pokud vstupní cesta obsahuje pouze disk (například C:) bez další cesty.

Výsledná cesta je poslána do jádra s požadavkem na změnu pracovního adresáře. Je-li cesta nevalidní, je z jádra vrácen chybový kód.

11.2 Příkaz md

Příkaz `md` slouží pro vytváření adresářů. Syntaxe příkazu je: `md [drive:]path`. Vstupní parametry jsou zpracovány pomocí regulárních výrazů. Nejprve jsou parametry rozděleny podle mezer. Cesta obsahující v názvu mezeru musí být obalena v jednoduchých nebo dvojitých uvozovkách. Každá cesta je poté rozdělena podle zpětných lomítek na adresáře. Nejprve se ověří, zda adresáře v dané cestě existují a že se jedná o adresáře. Ověření probíhá voláním jádra s požadavkem na otevření adresáře. Není-li adresář nalezen, je vytvořen dalším voláním jádra. Tento postup dvojího volání jádra je zvolen kvůli ověření, zda byl vytvořen alespoň jeden adresář. Existují-li všechny adresáře v cestě, je vypsána chybová hláška.

11.3 Příkaz rd

Příkaz `rd` slouží pro odstranění adresářů. Syntaxe příkazu je: `rd [/s] [/q] [drive:]path`. Parametr `/s` umožňuje smazat neprázdný adresář a parametr `/q` vypíná upozornění na smazání neprázdného adresáře. Nejprve jsou vstupní parametry zpracovány regulárním výrazem a poté jsou rozděleny podle mezer. Cesta obsahující v názvu mezeru musí být obalena v jednoduchých nebo dvojitých uvozovkách. Adresář, který se má odstranit je otevřen, aby se ověřilo, zda skutečně

existuje a jestli neobsahuje podadresáře a soubory. Je-li adresář prázdný, je smazán. Obsahuje-li další souboru a adresáře, je V závislosti na vstupních parametrech buď adresář a všechny podadresáře rekurzivně odstraněny nebo je vypsána hláška zda má být neprázdný adresář odstraněn. Otevření a odstranění adresáře je zprostředkováno voláním služeb jádra `Create_File` a `Delete_File`.

11.4 Příkaz dir

Příkaz `dir` vypíše obsah adresáře na výstup (`stdOutput`). Syntaxe příkazu je: `dir [/s] [drive:][path][dirname]`. Parametr `/s` určuje rekurzivní výpis, tedy výpis i všech podadresářů. Neobsahují-li vstupní argumenty žádnou cestu nebo název adresáře, je načten aktuální pracovní adresář pomocí služby jádra `Get_Current_Directory`. Na výstup (`stdOutput`) je vypsán název prohledávané složky. Adresář je otevřen službou `Create_File` a službou `Read` je získán obsah adresáře, který je okamžitě vypisován na výstup (`stdOutput`). Je-li nalezen adresář, a výpis má být rekurzivní, je adresář uložen do `vectoru` pro pozdější zpracování. Na konci, je-li výpis rekurzivní, je z `vectoru` vyjmut první adresář, který je stejným způsobem vypsán. Celý proces se opakuje dokud se neprojde celý adresářový strom.

11.5 Příkaz type

Příkaz `type` vypíše do konzole vstupní soubor nebo výstup z `stdInput`. Syntaxe příkazu je: `type [drive:][path]filename`. Obsahují-li argumenty klíčová slova:

con - jsou data načítána z `stdInput` a každý řádek je ihned vypsán zpět na `stdOutput`. Zadávání je ukončeno znakem `Ctrl+Z`.

nul - vypíše se nula znaků.

Neobsahují-li vstupní argumenty výše uvedená klíčová slova, jsou argumenty brány jako cesta k souboru, který se voláním služby jádra `Create_File` otevře a následně voláním služby jádra `Write` vypíše do `stdOutput`.

11.6 Příkaz sort

Příkaz `sort` seřadí abecedně řádky vstupního souboru nebo výstup z `stdInput`. Výsledek je vypsán do `stdOutput`. Syntaxe příkazu je: `sort [drive:][path]`. Jsou-li vstupní argumenty prázdné jsou data načítána z `stdInput`. V opačném případě je otevřen soubor podle vstupních argumentů a data jsou načítána ze souboru. Načítání probíhá voláním služby jádra `Read` kam je předán buďto handler otevřeného souboru, nebo `stdInput`. Čtení je ukončeno při chybě nebo při načtení nula znaků. Načítaná data jsou rozdělena po řádcích a vkládána do listu. Po ukončení načítání je list abecedně seřazen nehledě na velikost znaků. Nakonec je list vypsán pomocí služby jádra `Write` do `stdOutput`.

11.7 Příkaz echo

Příkaz echo přepošle vstupní argument na výstup zavoláním funkce `Write` z `rtl`.

11.8 Příkaz rgen

Generuje náhodná čísla v plovoucí čárce a ty vypisuje na `stdOutput`, dokud mu nepřijde znak `Ctrl+Z`. Program vytvoří vlákno pomocí funkce `Create_Thread` z `rtl`, která spustí funkci `wait_for_eof`. Funkce `wait_for_eof` čte z `stdInput`, dokud nepřečte znak `Ctrl+Z`. Po přečtení tohoto znaku se nastaví flag `generate` na `false`.

Po spuštění vlákna v hlavní funkci programu rgen se ve smyčce generují čísla a posílají se na výstup. Generování se ukončí po nastavení flagu `generate` na `false` ve funkci `wait_for_eof`.

11.9 Příkaz shell

Příkaz `shell` je simulací příkazové řádky s gramatikou cmd. Každý vytvoření shell umožňuje zpracovávat příkazy, které jsou zapsány do `stdInput` pomocí funkce z `Read_File` z `rtl` a potvrzeny klávesou `Enter`. Každá řádka je posléze samostatně zpracována. Ukončení příkazu je možné provést pomocí znaku `Ctrl+Z`.

`Shell` je i připraven zpracovat příkazy ze souboru a pomocí přesměrování `<` vykonávat tyto příkazy, které jsou zpracovány po řádcích dokud není přečten poslední z nich.

11.10 Příkaz ps

Příkaz `ps` vypíše všechny běžící procesy s jejich názvy, pidy a pidy rodičů. V první řadě se otevře adresář `0:\procfs` pomocí funkce `Create_File` z `rtl`. Následně z otevřeného souboru přečte obsah (informace o běžících procesech) a tento obsah zapíše na `stdOut` a uzavře otevřený soubor.

11.11 Příkaz shutdown

Příkaz `shutdown` ukončí celý systém. Příkaz zavolá metodu ze `shellu` `system_stop`. Tato metoda nastaví na `false` příznak `run_system`, který je společný pro všechny `shelly`. Zabrání se tak dalšímu čtení a dojde k jeho ukončení všech `shellů`.

11.12 Příkaz freq

Příkaz `freq` nemá žádné parametry je po jeho spuštění se čtou znaky z `stdInput` a sestaví frekvenční tabulku bytů. Po ukončení příkazu pomocí znaku `Ctrl+Z` nebo při chybě, jsou do `stdOut` vypsány takové byty, jejíž frekvence je větší než hodnota 0 a to v takovémto formátu: `0x%hhx : %d`, tj. znak je vypsán v hexadecimálním formátu.

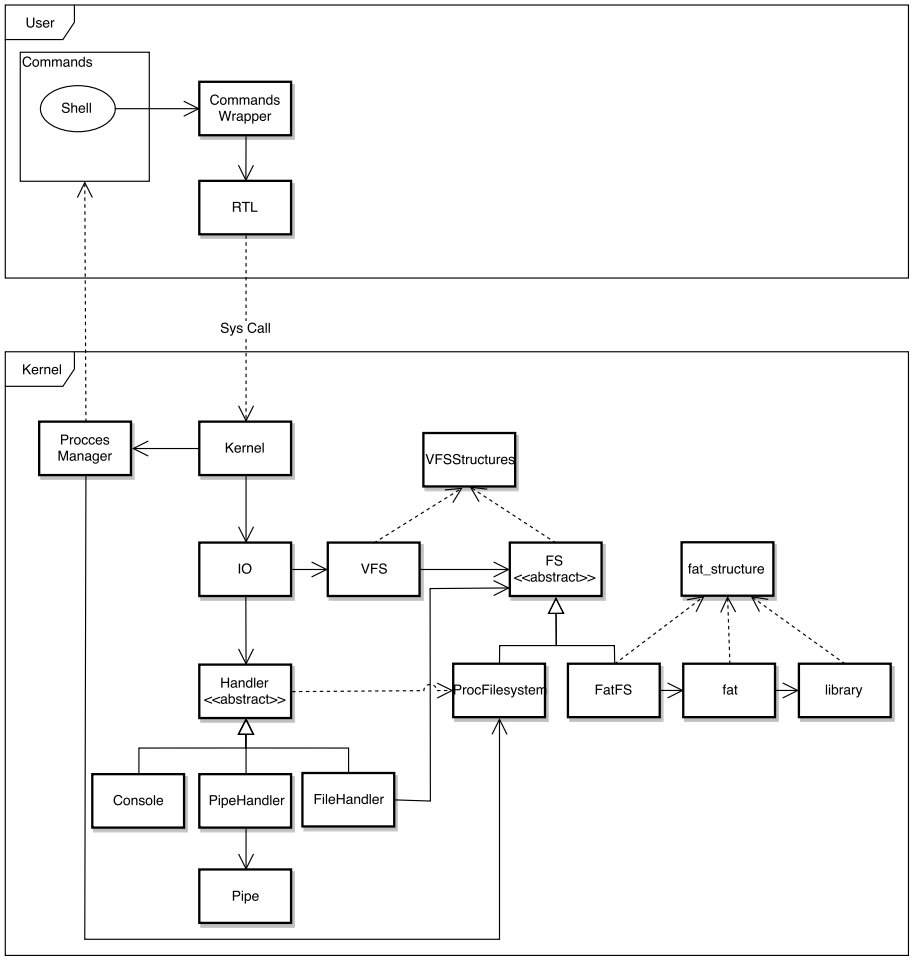
11.13 Příkaz wc

Příkaz `wc` vypíše do `stdOut` počet řádek, slov a znaků. Syntaxe příkazu je následující: `wc [drive:][path]filename`, kde počet vstupních suoborů je v rozsahu 0 až N. Není-li zadán ani jeden vstupní soubor, pak příkaz čte znaky přímo z `stdInput`, které jsou potvrzeny klávesou `Enter` a ukončení se provádí znakem `Ctrl+Z` nebo při chybě.

Pokud je jako vstup zadán alespoň jeden název souboru (či cesta), pak je z `rtl` zavolána funkce `Create_File` pro otevření existujícího souboru, jehož obsah je přečten a proveden příkaz `wc`, kde je i vypsán název. Pro více vstupních souborů je provedeno totéž a navíc je na konci vypsaný kompletní součet.

12 Celkový náhled na systém

Na obrázku 6 je možné vidět celkový pohled na systém, kde jsou znázorněny jednotlivé vazby.



Obrázek 6: Diagram systému.

13 Závěr

Veškerá funkčnost a příkazy popsány v zadání byly implementovány a řádně otestovány na vstup z konzole i na vstupní soubor s příkazy. Při implementaci byl použit a poupraven souborový systém `FAT` z předmětu `ZOS`. Program byl přeložen a všechna funkčnost vyzkoušena pro x64 a x86 v módu pro debug i release.