FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

KIV/OS - Semestrální práce

Simulátor operačního systému

student: Daniel HRYZBIL, Anežka JÁCHYMOVÁ, Zdeněk VALEŠ
datum: 29.11.2019

1 Zadání

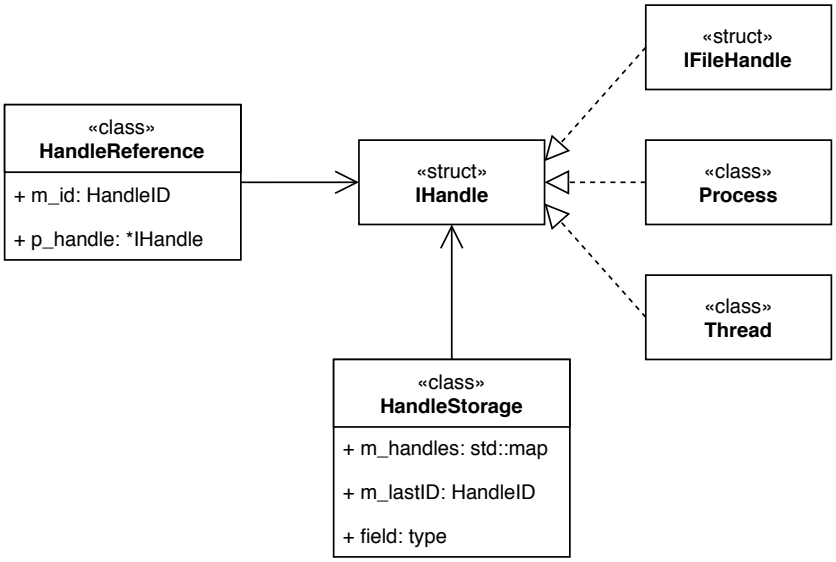
1. Vytvořte virtuální stroj, který bude simulovat OS
2. Součástí bude shell s gramatikou cmd, tj. včetně exit
3. Vytvoříte ekvivalenty standardních příkazů a programů:
   a) echo, cd, dir, md, rd, type, find /v /c""(tj. co dělá wc v unix-like prostředí), sort, tasklist, shutdown
      i. cd musí umět relativní cesty
      ii. echo musí umět @echo on a off
      iii. type musí umět vypsat jak stdin, tak musí umět vypsat soubor
   b) Dále vytvoříte programy rgen a freq
   c) rgen bude vypisovat náhodně vygenerovaná čísla v plovoucí čárce na stdout, dokud mu nepřijde znak Ctrl+Z //EOF
   d) freq bude číst z stdin a sestaví frekvenční tabulku bytů, kterou pak vypíše pro všechny byty s frekvencí větší než 0 ve formátu: "0x%hhx : %d"
4. Implementujte roury a přesměrování
5. Nebudete přistupovat na souborový systém, ale použijete simulovaný disk
   a) Za 5 bonusových bodů můžete k realizaci souborového systému použít semestrální práci z KIV/ZOS - tj. implementace FAT.

2 Kernel

Jádro poskytuje API pro správu procesů a vláken, vytvoření pipe a přístup k disku, ke kterému je možné z user space přistoupit systémovým voláním.

2.1 Handlery

Základním konceptem kernelu je třída `HandleStorage` společně s `HandleReference`. Každá třída reprezentující nějaký handle (soubor, proces, vlákno, ...) implementuje rozhraní `IHandle`. Všechny instance těchto tříd jsou následně uloženy uvnitř `HandleStorage`, kde jim je přiřazeno jejich unikátní `HandleID` (16 bitové číslo). Přístup k uloženým handle je následně možný pouze přes objekty `HandleReference`. Struktura handlerů je znázorněna diagramem tříd na obrázku 1.



Obrázek 1: Diagram tříd handlerů

Uvnitř `HandleStorage` je pro každý handle uložen počet referencí na tento handle, který se inkrementuje vždy při vytvoření nové `HandleReference` a dekrementuje při odstranění `HandleReference`. Handly s nulovým počtem referencí jsou automaticky uzavřeny a odstraněny ze systému.

Každý proces si udržuje vlastní množinu referencí na handly, které byly v jeho kontextu vytvořeny nebo jinak získány (například při vytváření procesu předá rodičovský proces nově vytvořenému procesu handle na `stdin` a `stdout`). Tato množina se zároveň používá i pro zjištění, zda má proces k nějakému handlu vůbec přístup. Při odstranění nějakého procesu potom dojde k odstranění všech handle referencí uložených v jeho množině, a tím dojde automaticky i k odstranění všech handlů, které používal pouze daný proces.

Kernel se tak nespoléhá na "slušnost"user-space kódu, který by měl vždy použít `CloseHandle`, ale dokáže při ukončení procesu automaticky uklidit všechny nepotřebné handly stejně jako reálný OS. Zároveň tento koncept umožňuje i efektivnější spolupráci více vláken najednou. Při práci s nějakým handle totiž není potřeba zamykat globální zámky, protože pokud pro tento handle existuje alespoň jeden objekt `HandleReference`, tak je zaručeno, že žádné jiné vlákno nemůže tento handle nečekaně odstranit a způsobit tak pád celého systému.

2.2 Procesy a vlákna

Procesy jsou uloženy v již zmíněném `HandleStorage` a `HandleID` je bráno jako PID. Stejně tak jsou uložena i vlákna a `HandleID` představuje TID.

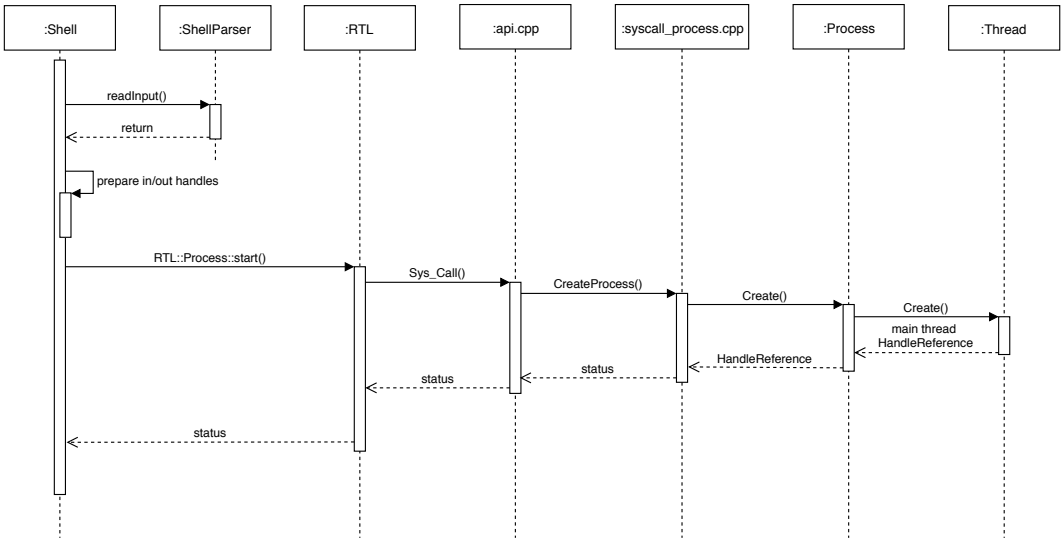
Každé vlákno simulovaného OS využívá reálné vlákno fyzického OS (`std::thread`). Kernel dále neobsahuje žádný platform-specific kód, kromě načítání "user.dll". Za hlavní vlákno procesu se považuje jeho první vlákno. Při vytvoření procesu je vždy vytvořeno

i jeho hlavní vlákno.

2.2.1 API

Procesy a vlákna jsou reprezentovány handly, API pro práci s nimi tedy očekává (případně vrací) patřičný handle.

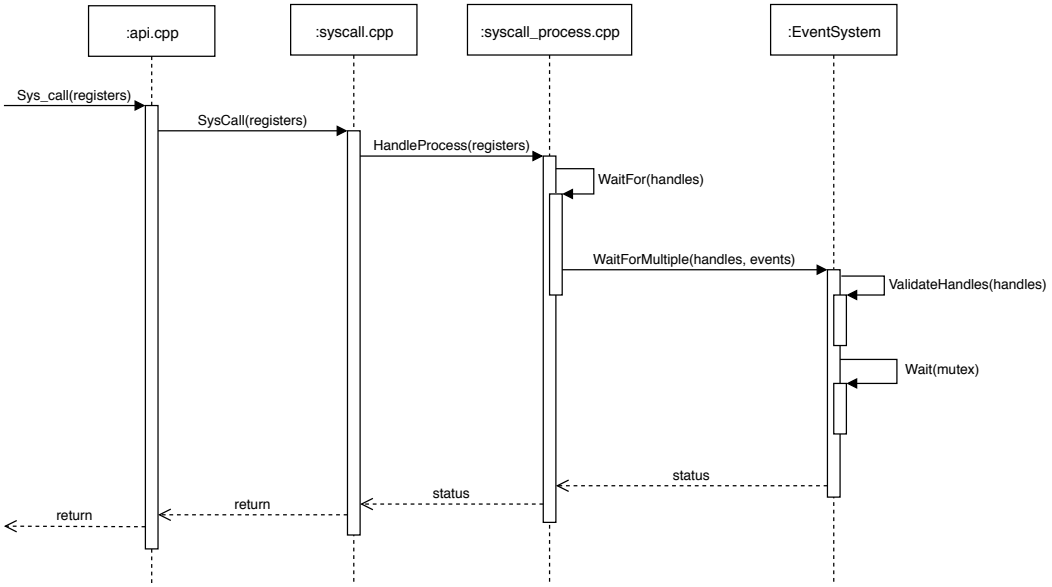
Create Process Vytvoří nový proces.



Obrázek 2: Sekvenční diagram vytvoření procesu

Create Thread Vytvoří nové vlákno pro aktuální proces.

Wait For Počká na konec procesu, nebo vlákna.



Obrázek 3: Sekvenční diagram čekání na vlákno, nebo proces

Get Exit Code Získá exit kód procesu, nebo vlákna.

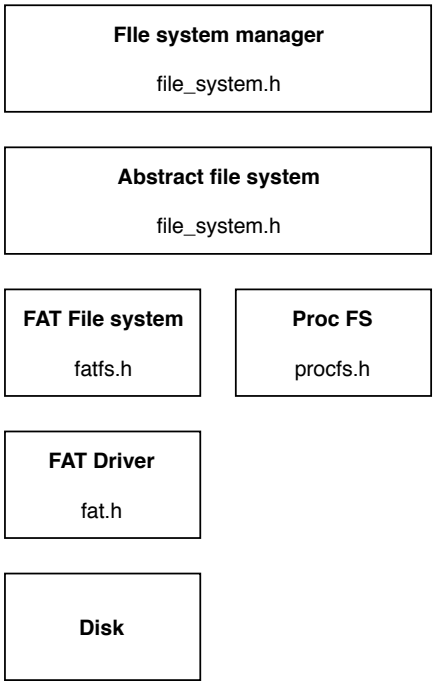
Setup Signal Nastaví signal handler.

Exit Nastaví exit kód aktuálímu vláknu.

System Shutdown Pošle signál terminate na všechna vlákna v systému.

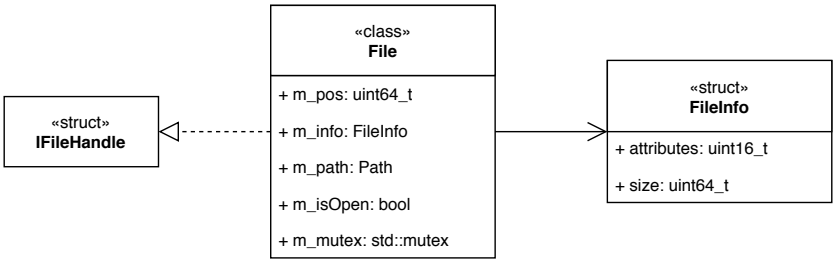
2.3 Filesystem

Kernel obsahuje file system manager, který spravuje disky a souborové systémy na nich uložené. Pro přístup na disk je použito abstraktní API, které implementuje každý ovladač pro FS. Struktura je znázorněna na obrázku 4.



Obrázek 4: Struktura správy souborového systému

Soubory Handly otevřených souborů jsou uloženy ve výše zmíněném `HandleStorage` – třída `File`, která představuje handle otevřeného souboru, implementuje `IFileHandle`. K synchronizaci přístupu více vlákny je použit mutex (`std::mutex`). Struktura handleru je znázorněna na obrázku 5.



Obrázek 5: Struktura souborového handleru

2.3.1 API

K datům každého souborového systému je přistupováno skrze API definované v `IFileSystem`. Uživatelská vrstva pak k API přistupuje skrze systémová volání.

Create Vytvoří soubor.

Query Získá info o souboru.

Read Přečte soubor (nebo jeho část) do bufferu. Offset od kterého se čte je získán z handleru souboru.

ReadDir Načte soubory v adresáři.

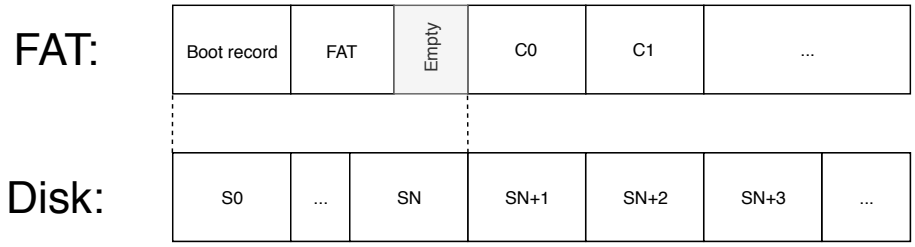
Remove Odstraní soubor, nebo prázdný adresář.

Resize Změní velikost souboru.

Write Zapíše data do souboru. Offset od kterého se zapisuje je získán z handleru souboru.

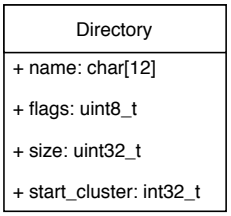
2.3.2 FAT

Jako hlavní souborový systém jsme zvolili FAT implementovanou v rámci KIV/ZOS. Rozložení souborového systému na disku je znázorněno na obrázku 6. V původní specifikaci je na disku uložena alespoň jedna další kopie FAT tabulky, která se používá ke kontrole bloků dat. Protože kontrolu poškozených dat v práci neprovádíme, rozhodli jsme se ukládat pouze jednu kopii.



Obrázek 6: Uložení FAT souborového systému na disk

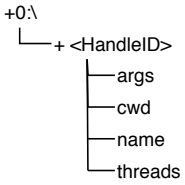
Velikost jednoho datového clusteru je dána velikostí sektorů na disku a platí, že 1 cluster = N $\times$ sektor, kde N je vypočítané při inicializaci souborového systému na disku. Root adresář začíná na clusteru 0. Metadata každého souboru jsou reprezentována strukturou `Directory` (obrázek 7). Adresáře jsou od souborů odlišeny nastaveným bitem ve `flags`.



Obrázek 7: Struktura Directory

2.3.3 ProcFS

Přístup k aktuálně běžícím procesům je realizován přes virtuální souborový systém, který je mapován na disk `0:\`. Struktura je znázorněna na 8. V kořenovém adresáři existuje pro každý běžící proces adresář pojmenovaný podle `HandleID` daného procesu. Adresář pak obsahuje popisné soubory, ze kterých je možné o procesu přečíst: parametry se kterými byl spuštěn, aktuální pracovní adresář, jméno a počet běžících vláken.



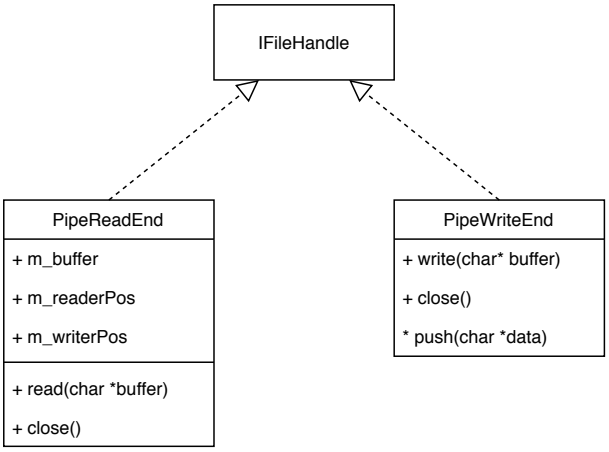
Obrázek 8: Struktura ProcFS

`ProcFS` dědí od `IFileSystem` a implementuje tedy celé API pro přístup k datům. Všechny funkce kromě `query()`, `read()` a `readDir()` vrací chybu `PERMISSION_DENIED`. K vypsání všech běžících procesů lze použít funkce `readDir(0:\)`.

Detaily libovolného procesu je možné získat voláním `read(0:\<HandleID>\<proc_file_name>)`. Místo `HandleID` lze použít `self`, API pak vrací detaily o aktuálně běžícím procesu.

2.4 Roura

Roura je tvořena dvěma propojenými handly - `PipeReadEnd` a `PipeWriteEnd` viz obrázek 9. Čtecí konec obsahuje buffer, do kterého se zapisuje ze zapisovacího konce (metoda `push()`). Při uzavření jednoho konce se spojení mezi nimi přeruší a druhý konec tak ví, že už při prázdném bufferu nemá čekat na další data (čtecí konec) nebo při plném bufferu čekat na uvolnění místa (zapisovací konec).



Obrázek 9: Pipe diagram tříd

3 RTL

Vrstva mezi uživatelskými programy a kernelem (nečekaně). Wrappery pro systémová volání. Datová reprezentace souborů a procesů pro user space. Překlad chybových stavů na text.

TODO: něco napsat sem? Asi o práci se vstupem/výstupem

4 Shell a uživatelské příkazy

Proces shellu je spuštěn po startu systému. Shell ve smyčce čte po řádcích ze standardního vstupu, řádky parsuje a spouští rozparsované příkazy. Pokud shell obdrží signál `TERMINATE`, smyčka je ukončena a shell se vypne.

4.1 Parser

TODO: informace o parseru Implementován třídou `ShellParser` v `shell.cpp`. Počítá s řádkem o max délce 4096. Z rozparsované řádky vrátí strukturu obsahující příkazy, které se mají vykonat a případné přesměrování mezi nimi (roura, `>`, `>>`).

4.2 Implementace příkazů

Každý příkaz je shellem spouštěn jako samostatný proces. Před spuštěním shell procesu nastaví handlery na vstup a výstup (stdIn/Out, roura, nebo soubory). Kód příkazu pak volá funkce RTL pro čtení a zápis. RTL si z kontextu aktuálně běžícího vlákna sama získá tyto handlery, takže kód příkazu neví kam zapisuje, nebo odkud čte.

Přesměrování `>` nastaví velikost cílového souboru na 0 a pak do něj od začátku zapisuje. Přesměrování `>>` nastaví pozici cílového souboru na jeho konec a pak za něj zapisuje data.

dir Vypíše na výstup seznam souborů v zadaném adresáři. Cest lze příkazu skrze parametry předat více. Příkaz otevře zadaný adresář funkcí `OpenDirectory(path)` z RTL. Obsah adresáře je získán RTL funkcí `GetDirectoryContent(dirHandle)`.

echo Vypíše text ze vstupu na výstup. Echo lze vypnout příkazem `@echo off` (a zapnout `@echo on`). Proměnná, která drží stav echa je uložena v každé instanci shellu zvlášť. K výpisu na výstup je použita funkce RTL `WriteFile(handle)`.

find Podporujeme pouze příkaz `find /C /V ""`, tj. s prázdným řetězcem k vyhledání. Příkaz čte po řádkách text ze vstupu dokud nepřijde EOF. Pak vypíše počet zadaných řádek. Význam přepínačů:

- `/C` Počet řádek, které text obsahují,
- `/V` Zobrazí řádky, které zadaný text neobsahují.

freq Čte vstup dokud nepřijde EOF a udělá statistiku, kolikrát byl který byte na vstupu přítomen. Po dokončení statistiky je výsledek vypsán na výstup.

md Vytvoří zadaný adresář. Adresářů lze skrze parametry předat více. Příkaz vytvoří adresář voláním funkce RTL `CreateDirectory(path)`.

rd Smaže zadaný soubor. Použitím parametru `/S` lze rekurzivně mazat adresářové stromy. Soubor je smazán voláním funkce RTL `DeleteDirectory(path)`.

rgen Generuje náhodná čísla v rozmezí $< 0; 1)$, která vypisuje na výstup. Čísla jsou generována dokud na vstupu není EOF.

shutdown Vypne systém voláním funkce RTL `Shutdown()`.

sort Čte řádky ze vstupu a řadí je. Po přečtení EOF vypíše seřazené řádky na výstup.

tasklist Zavolá `OpenDirectory("0:")` čímž otevře vituální disk 0 (viz sekce 2.3.3), načte jeho obsah (`GetDirectoryContent()`) a vypíše jej na výstup.

type Vypíše na výstup obsah zadaných souborů. K otevření souboru je použita funkce RTL `OpenFile(path)`, k získání obsahu pak `ReadFile(handle, buffer)`.

5 Závěr

Simulátor operačního systému se nám podařilo úspěšně implementovat.
TODO: co bychom mohli příště udělat líp
TODO: co bylo nejzajímavější