FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

KIV/OS - Semestrální práce

Simulátor operačního systému

autoři: Daniel HRYZBIL (A19N0076P),
Anežka JÁCHYMOVÁ (A19N0077P),
Zdeněk VALEŠ (A17N0094P)

datum: 29.11.2019

1 Zadání

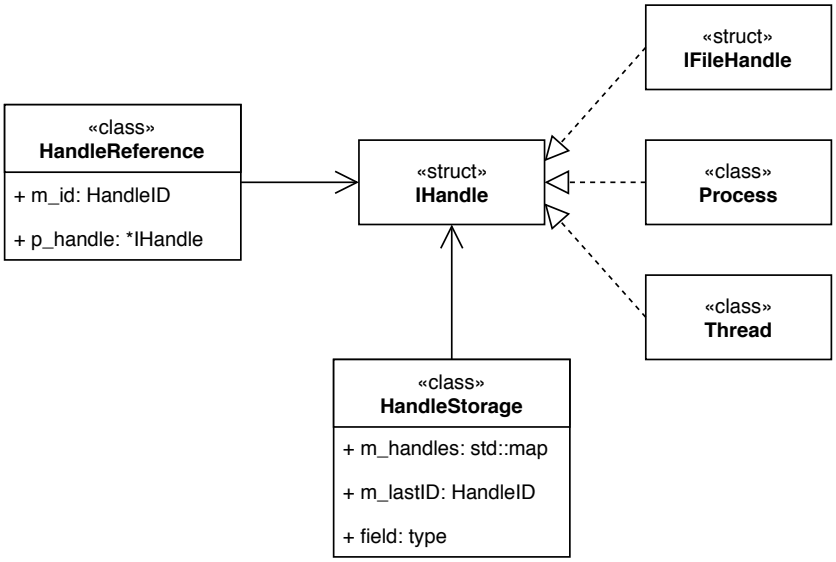
1. Vytvořte virtuální stroj, který bude simulovat OS
2. Součástí bude shell s gramatikou cmd, tj. včetně exit
3. Vytvoříte ekvivalenty standardních příkazů a programů:
   a) echo, cd, dir, md, rd, type, find /v /c""(tj. co dělá wc v unix-like prostředí), sort, tasklist, shutdown
      - cd musí umět relativní cesty
      - echo musí umět @echo on a off
      - 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
   - 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 Handle

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). Toto číslo se také předává do user-space jako indentifikátor daného handle. Přístup k uloženým handle je následně možný pouze přes objekty `HandleReference`. Struktura handlů je znázorněna diagramem tříd na obrázku 1.



Obrázek 1: Diagram tříd handle

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 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.

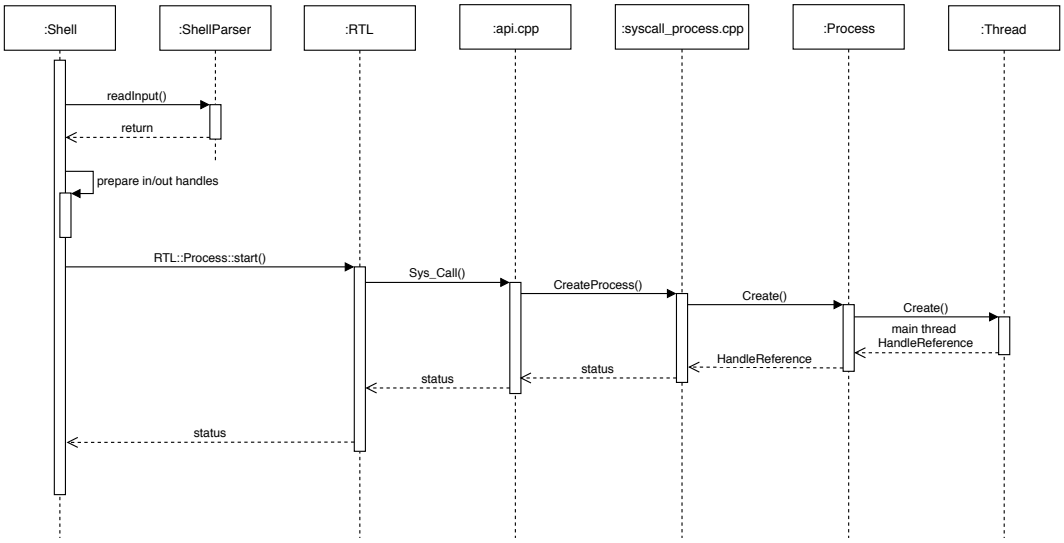
2.2 Procesy a vlákna

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`). 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.

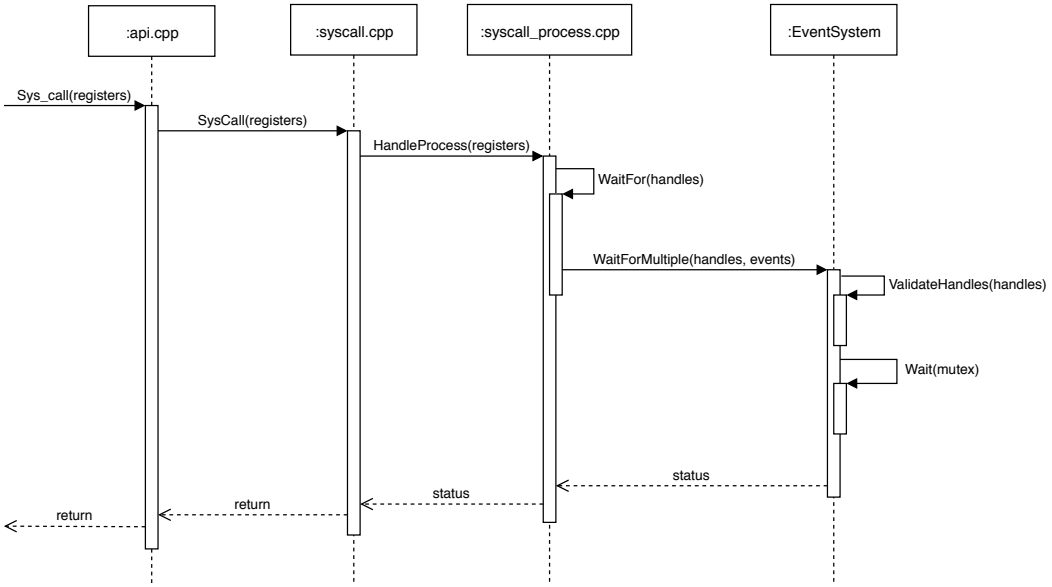
Create Process Vytvoří nový proces.



Obrázek 2: Sekvenční diagram vytvoření procesu

Create Thread Vytvoří nové vlákno v kontextu aktuálního procesu.

Wait For Počká, až se zadané procesy nebo vlákna ukončí.



Obrázek 3: Sekvenční diagram čekání na vlákno nebo proces

Get Exit Code Získá návratový kód procesu nebo vlákna.

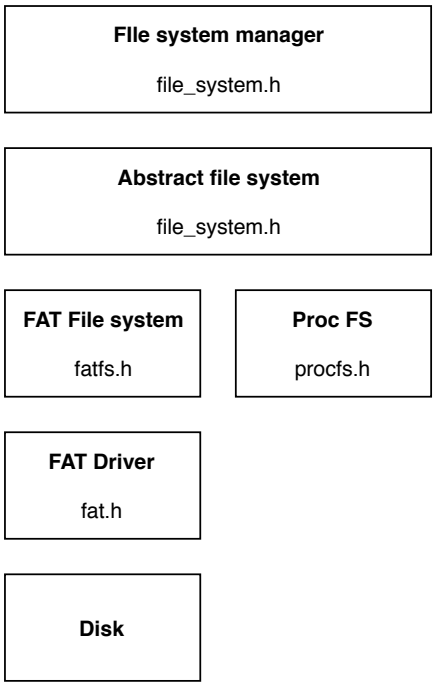
Setup Signal Nastaví signal handler. Signály se posílají vždy vláknům. Každé vlákno má svůj signal handler a seznam signálů (signal mask), které se mohou doručit tomuto handleru. Pro jednoduchost má každé vlákno pouze jeden signal handler společný pro všechny signály (v OS je definovaný stejně jen jeden signál). Při každém vstupu vlákna do jádra (syscall) se zkontroluje, zda nebyl danému vláknu odeslán nějaký signál, a pokud ano, tak se spustí jeho signal handler (pokud nějaký má a pokud je daný signál v seznamu signálů, které se mohou doručit). Stejná kontrola se provádí i při výstupu vlákna z jádra (konec syscallu). To kvůli delšímu blokování vlákna v jádře (například čekání na vstup od uživatele nebo čekání na ukončení jiného procesu nebo vlákna). Signal handler se tak spouští vždy v kontextu vlákna, ve kterém byl nastaven.

Exit Nastaví návratový kód aktuálního vlákna. Samotné ukončení musí vlákno provést samo návratem ze své vstupní funkce. Standardní knihovna C++ ani neposkytuje žádný způsob, jak ukončit běžící vlákno (`std::thread`). Bylo by sice možné použít platform-specific funkce z WinAPI (`ExitThread` nebo `TerminateThread`), ale násilné ukončení běžícího vlákna představuje z pohledu jazyka C++ velký problém, protože se nemusejí zavolat všechny destruktory a nedojde tak k uvolnění všech zdrojů alokovaných v daném vlákně. Z tohoto důvodu se vlákno musí vždy ukončit samo, což by v reálném OS nebylo moc dobré. Návratový kód procesu je vždy návratový kód hlavního vlákna daného procesu. Proces se považuje za ukončený, pokud už neobsahuje žádná běžící vlákna.

System Shutdown Uzavře všechny souborové handly a pošle signál `Terminate` všem aktivním vláknům v systému.

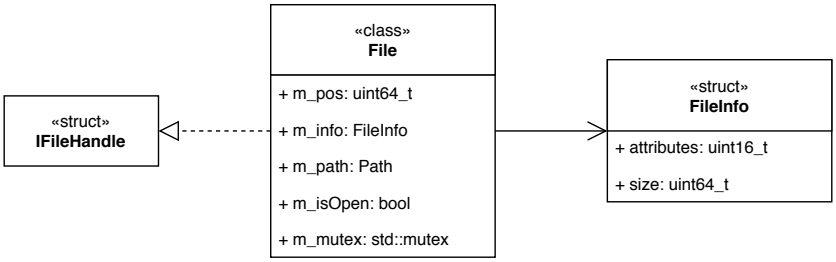
2.3 Filesystem

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



Obrázek 4: Struktura správy souborových systémů

Soubory Každý souborový handle implementuje rozhraní `IFileHandle`. Handly otevřených souborů a adresářů jsou uloženy ve výše zmíněném `HandleStorage`. Otevřený soubor nebo adresář je reprezentován objektem třídy `File`, která také implementuje `IFileHandle`. K synchronizaci přístupu více vlákny je použit mutex (`std::mutex`). Struktura handlu je znázorněna na obrázku 5.



Obrázek 5: Struktura souborového handle

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 nebo adresář.

Query Získá informace o souboru nebo adresáři.

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

ReadDir Načte soubory v adresáři.

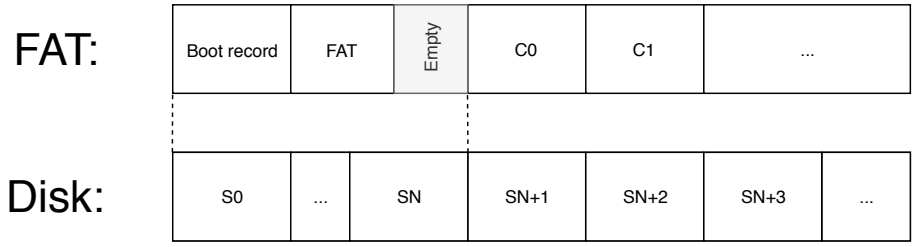
Remove Odstraní soubor nebo prázdný adresář.

Resize Změní velikost souboru.

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

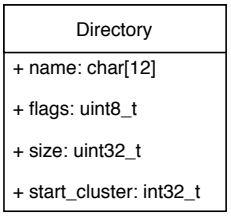
2.3.2 FAT

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 × 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`. Jméno souboru může být maximálně 11 znaků (12. znak je `'\0'`).

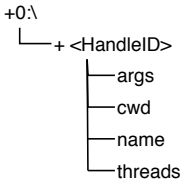


Obrázek 7: Struktura Directory

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. Dále je zde adresář `self`, který vždy odkazuje na aktuální proces. Adresáře jednotlivých procesů obsahují popisné soubory, ze kterých je možné přečíst informace o daném procesu:

- Název programu
- Parametry se kterými byl proces spuštěn
- Aktuální pracovní adresář
- Počet běžících vláken



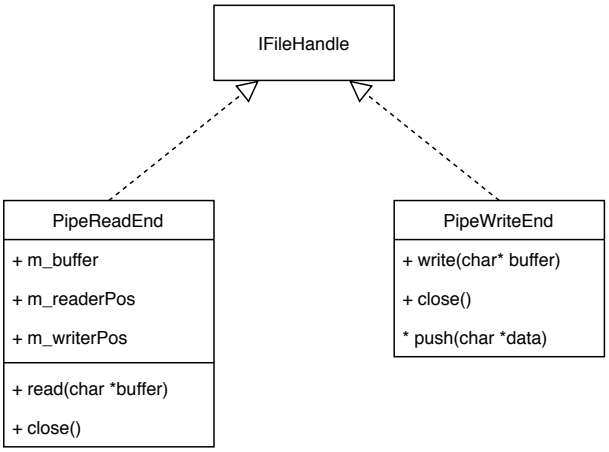
Obrázek 8: Struktura ProcFS

`ProcFS` také dědí od `IFileSystem` a implementuje tedy kompletní API pro přístup k datům. Celý `ProcFS` je ale pouze pro čtení, takže všechny funkce kromě `query()`, `read()` a `readDir()` vždy vrací chybu `PERMISSION_DENIED`. K vypsání všech běžících procesů lze použít funkci `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. 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

Runtime knihovna obaluje systémová volání a umožňuje tak pohodlnou komunikaci user-space programů s jádrem. Obsahuje také obalové třídy reprezentující proces, vlákno, soubor, adresář a rouru. Celá se nachází v `rtl.h` a `rtl.cpp`.

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

Implementován třídou `ShellParser` v `shell.cpp`. Z rozparsované řádky vrátí strukturu obsahující příkazy, které se mají vykonat a informace o případném přesměrování (`<`, `>`, `>>`). 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.

4.2 Implementace příkazů

Každý příkaz je shellem spouštěn jako samostatný proces. Před spuštěním shell procesu nastaví handly na standardní vstup a výstup. Kód příkazu pak volá funkce RTL pro čtení ze standardního vstupu a zápis do standardního výstupu. RTL si z kontextu aktuálně běžícího vlákna sama získá handly standardního vstupu a výstupu daného procesu, takže získání a uložení těchto handlů není potřeba řešit v kódu každého programu.

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

find Podporujeme pouze příkaz `find /C /V ""`, tj. s prázdným řetězcem k vyhledání. Příkaz čte po řádcích 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 virtuální 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 `RTL::ReadFile(handle, buffer)`.