



KIV/OS - Semestrální práce

Simulátor operačního systému

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

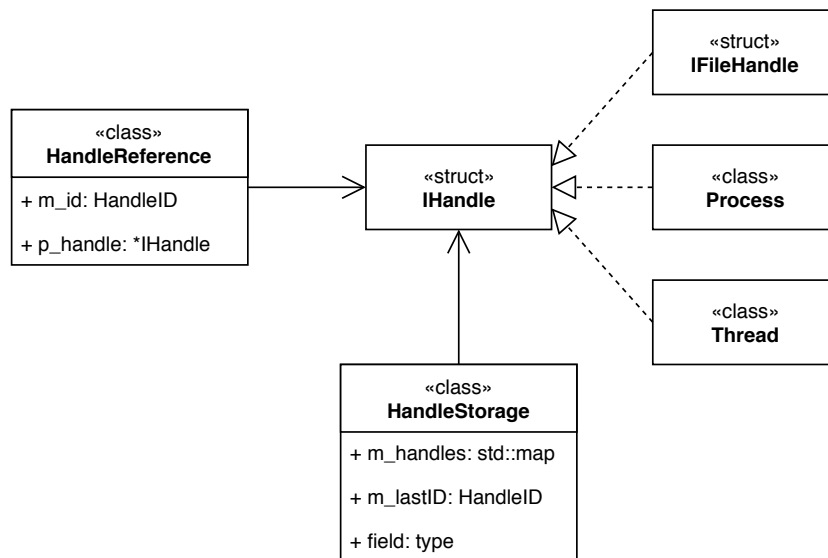
1 Zadání

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. TODO: trochu rozvést

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 uložený 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 nespolehá 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.

2.1 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.

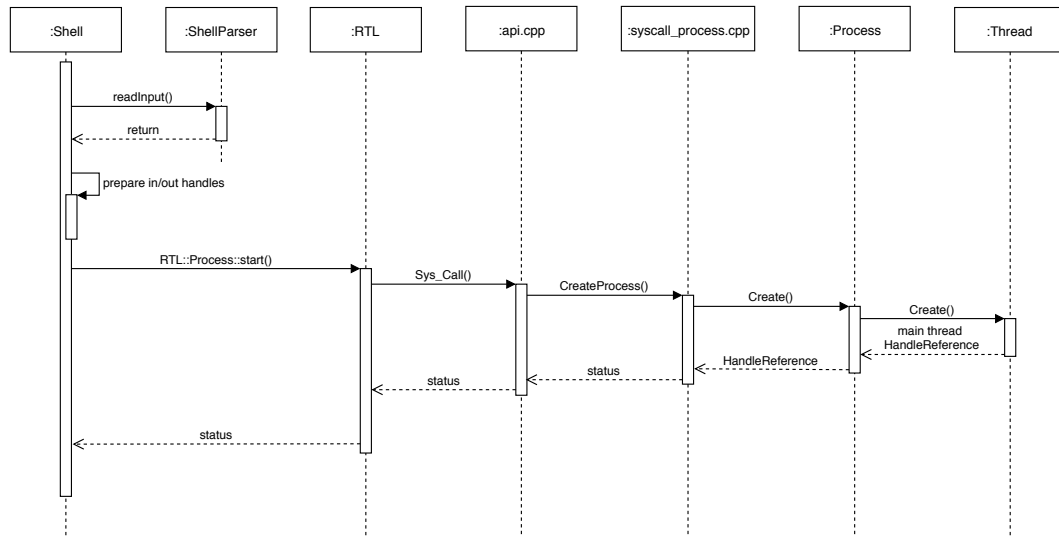
Každé vlákno simulovaného OS využívá reálné vlákno fyzického OS (`std::thread`). Kernel dále neobsahuje žádný platform-specific kód, kromě načítání "user.dll". Za hlavní vlákno procesu se považuje jeho první vlákno. Při vytvoření procesu je vždy vytvořeno

i jeho hlavní vlákno.

2.1.1 API

K API je přístupováno skrze systémová volání.

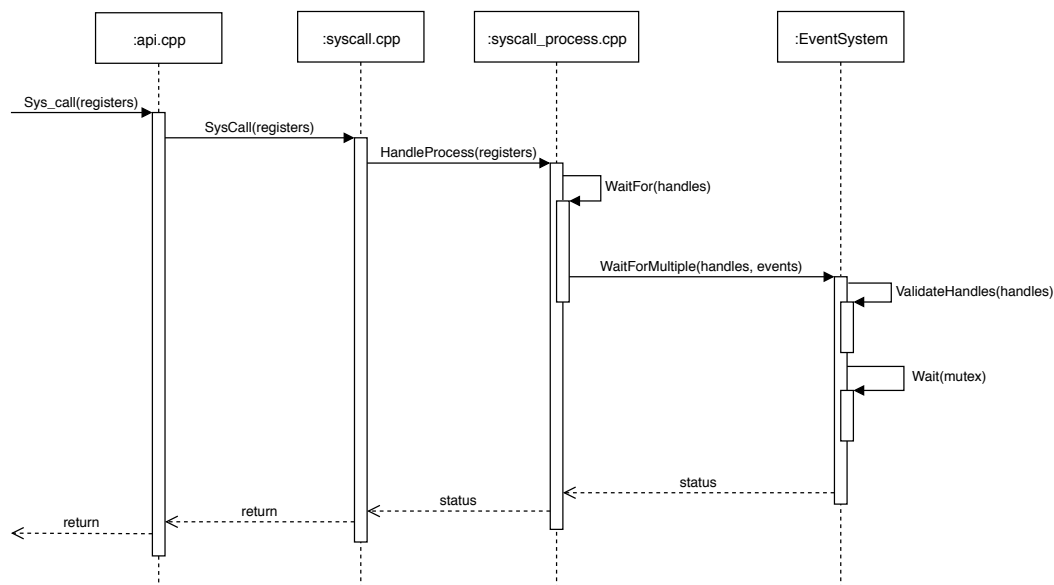
Create Process Vytvoří nový proces.



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.

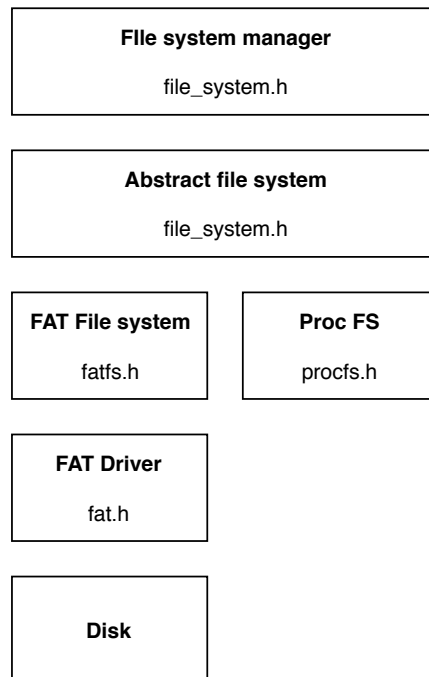
Setup Signal Nastaví signal handler.

Exit Nastaví exit kód aktuálnímu vláknu.

System Shutdown Pošle signál terminate na všechna vlákna v systému.

2.2 Filesystem

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



Obrázek 4: Struktura správy souborového systému

Otevřené soubory Handly uložených souborů jsou uloženy ve výše zmíněném `HandleStorage` – třída `File`, která představuje handle otevřeného souboru, dědí od `IFileHandle`. K synchronizaci přístupu více vláken je použit mutex (`std::mutex`).

2.2.1 API

K datům každého souborového systému je přístupováno 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.

ReadDir Načte soubory v adresáři.

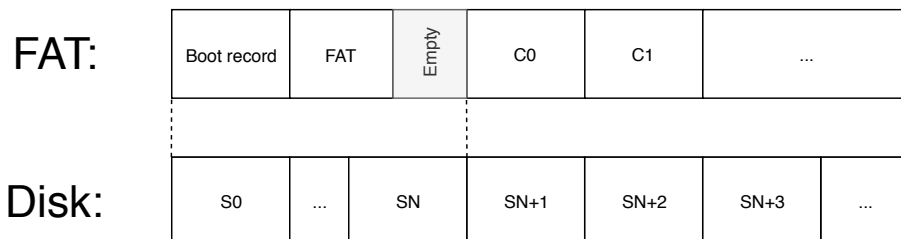
Remove Odstraní soubor, nebo adresář.

Resize Změní velikost souboru.

Write Zapiše data do souboru.

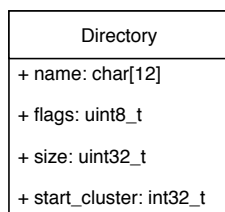
2.2.2 FAT

Jako filesystem jsme zvolili FAT implementovanou v rámci KIV/ZOS. Rozložení souborového systému na disku je znázorněno na obrázku 5. 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 5: Uložení FAT souborového systému na disk

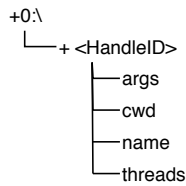
Velikost jednoho datového clusteru je dána velikostí sektorů na disku a platí, že $1 \text{ cluster} = N \times \text{sektor}$, kde N je vypočítané při inicializaci souborového systému na disku. Root adresář začíná na clusteru 0. Metadata každého souboru jsou reprezentována strukturou `Directory` (obrázek 6). Adresáře jsou od souborů odlišeny nastaveným bitem ve `flags`.



Obrázek 6: Struktura Directory

2.2.3 ProcFS

Přístup k aktuálně běžícím procesům je realizován přes virtuální souborový systém, který je mapován na disk `0:\`. Struktura je znázorněna na 7. 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 přejít: parametry se kterými byl spuštěn, aktuální pracovní adresář, jméno a počet běžících vláken.



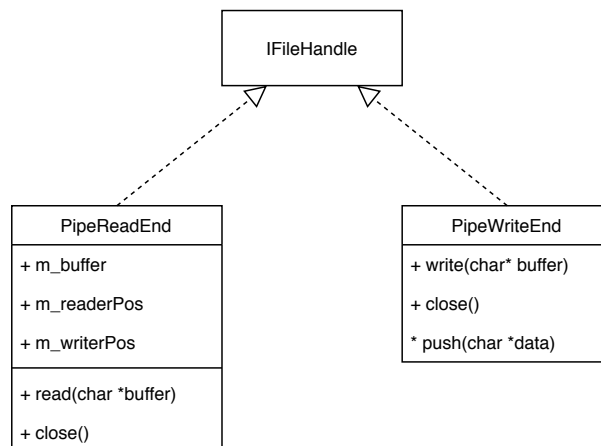
Obrázek 7: Struktura ProcFS

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.

2.3 Roura

Roura je tvořena dvěma propojenými handly - `PipeReadEnd` a `PipeWriteEnd` viz obrázek 8. 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 8: Pipe diagram tříd

3 RTL

Vrstva mezi uživatelskými programy a kernelem (nečekaně).

TODO: něco napsat sem?

4 Shell a uživatelské příkazy

4.1 Parser

TODO: informace o parseru

4.2 Implementace příkazů

TODO: nějaký krátký info k příkazům

dir

echo

find

freq

md

rd

rgen

shutdown

sort

tasklist

type

5 Závěr

Simulátor operačního systému se nám podařilo úspěšně implementovat.

TODO: co bychom mohli příště udělat líp

TODO: co bylo nejzajímavější