



Dokumentace k semestrální práci z KIV/OS

Simulace operačního systému

Studenti: Lukáš Černý, Petr Volf, Lenka Šimečková
St. čísla: A17N0065P, A18N0103P, A18N0101P
E-maily: luccerny@students.zcu.cz, volfpe@students.zcu.cz, simeckol@students.zcu.cz
Datum: 26.11.2018

Obsah

1	Zadání	1
2	Procesy	3
2.1	Reprezentace procesu	3
2.2	Reprezentace vlákna	4
2.3	Správa procesů	5
2.3.1	Synchnorizace procesů	5
3	IO	7
3.1	Souborový systém (FAT16)	8
3.1.1	Boot sector	9
3.1.2	Entire Directory	10
3.2	Přesměrování a roury	11
4	Shell	13
5	Uživatelské funkce	15
5.1	echo	15
5.2	cd	15
5.3	dir	16
5.4	md	16
5.5	rd	17
5.6	type	17
5.7	find /v /c""	17
5.8	sort	18
5.9	tasklist	18
5.10	shutdown	19
5.11	rgen	19
5.12	freq	19
6	Závěr	21

1 Zadání

- Vytvořte virtuální stroj, který bude simulovat OS
- Součástí bude shell s gramatikou `cmd`
- Vytvoříte ekvivalenty standardních příkazů a programů
 - `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
 - Dále vytvoříte programy `rgen` a `freq`
 - `rgen` bude vypisovat náhodně vygenerovaná čísla v plovoucí čárce na `stdout`, dokud mu nepřijde znak `Ctrl+Z` //EOF
 - `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"`
- Implementujte roury a přesměrování
- 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.

Při zpracování tohoto zadání použijte a dále pracujte s kostrou tohoto řešení, kterou najdete v archívu `os_simulator.zip`. Součástí archívu, ve složce `compiled`, je soubory `checker.exe` a `test.exe`. Soubor `checker.exe` je validátor semestrálních prací. Soubor `test.exe` generuje možný testovací vstup pro vaši semestrální práci.

Vaše vypracování si před odevzdáním zkontrolujte programem `checker.exe`. V souboru `checker.ini` si upravte položku `Setup_Environment_Command`, v sekci General, tak, aby obsahovala cestu dle vaší instalace Visual Studia. Např. vzorové odevzdání otestujete příkazem `"compiled`

vzorove odevzdani.zip”, spuštěného v kořenovém adresáři rozbaleného archívu. Odevzdávaný archív nemá obsahovat žádné soubory navíc a program musí úspěšně proběhnout.

Další informace budou poskytnuty na seminářích a přednáškách dle potřeby.

2 Procesy

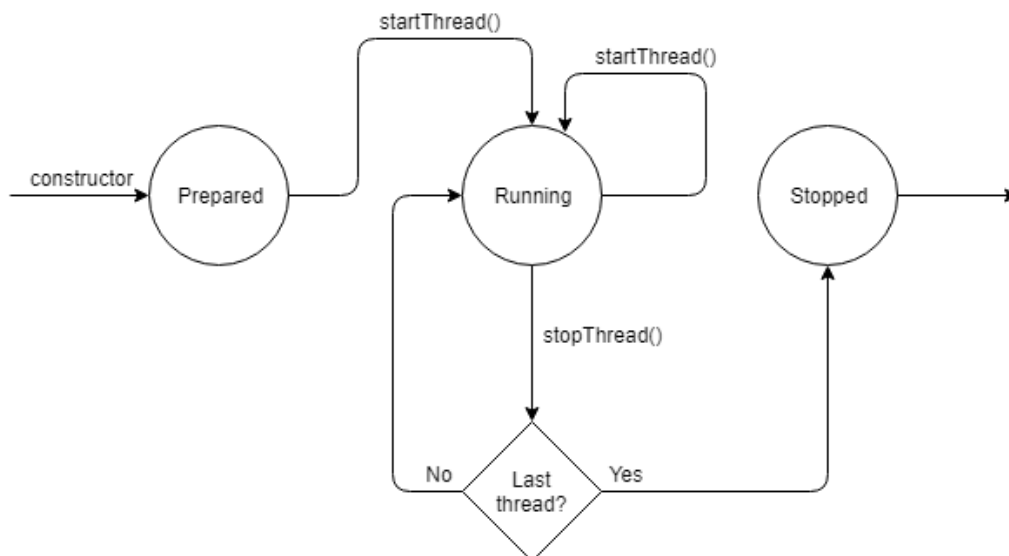
Proces je instance počítačového programu. Informace o procesu jsou v operačním systému uloženy ve struktuře PCB, která uchovává jeho identifikátor (PID), identifikátor rodiče, stav (běžící/ukončený..) a další informace.

2.1 Reprezentace procesu

Proces je v semestrální práci realizovaný instancí třídy `Process`, která představuje analogii k záznamu PCB v reálném operačním systému. Obsah třídy `Process` není identický se záznamem PCB, jelikož v semestrální práci některé atributy této struktury nejsou potřeba. Tato třída obsahuje následující atributy:

- **pid**: Identifikátor procesu (hostujícího OS).
- **parent_pid**: Identifikátor rodičovského procesu (hostujícího OS).
- **handle**: Identifikátor procesu našeho OS.
- **parent_handle**: Identifikátor rodičovského procesu našeho OS.
- **state**: Stav procesu. Může nabývat následujících stavů:
 - **Prepared**.
 - **Running**.
 - **Stopped**.
- **userfunc_name**: Název uživatelského procesu (název funkce v dynamické knihovně).
- **working_dir**: Pracovní adresář procesu.
- **threads**: Mapa vláken procesu.

Třída `Process` poskytuje metody pro vytvoření vlákna, ukončení vlákna a odstranění vlákna ze seznamu vláken procesu (po přečtení návratového kódu vlákna). Při vytvoření a spuštění prvního vlákna se změní stav procesu z `Prepared` na `Running` a PID procesu se nastaví na TID tohoto vlákna. Po ukočení posledního vlákna se stav procesu změní na `Stopped`. Stavový diagram procesu lze vidět na obrázku 2.1.



Obrázek 2.1: Stavový diagram procesu.

2.2 Reprezentace vlákna

Vlákno je stejně jako proces reprezentováno třídou a je analogií k TCB v operačním systému. Třída vlákna (**Thread**) obsahuje následující atributy:

- **tid**: Identifiátor vlákna (hostujícího OS).
- **state**: Stav vlákna (**Prepared**, **Running**, **Stopped**).
- **thread_obj**: Instance `std::thread` reprezentující vlákno v hostujícím OS.
- **func_addr**: Adresa vstupního bodu funkce vykonávaného vlákna.
- **handlers**: Mapa handlerů pro obsluhu signálů.
- **context**: Registry vlákna.

Třída vlákna poskytuje metody pro start a ukočení vlákna. Při ukončení vlákna se volá funkce `TerminateThread` z *Windows API*, aby se zajišťovalo opravdové ukončení běžícího procesu na hostitelském OS. Životní cyklus vlákna lze vidět na obrázku 2.2.



Obrázek 2.2: Stavový diagram vlákna.

2.3 Správa procesů

Správa procesů je řešená třídou `Process_Manager`. Tato třída obsluhuje systémové volání, které se týkají procesů. Obslužné funkce systémových volání implementují API, které je součástí kostry semestrální práce.

Handle procesu/vlákna je inkrementující se číslo (od čísla 1). Každé nové vlákno dostane další číslo v pořadí. Správce procesů obsahuje mapu, do které si ukládá pár *handle* - *nativní thread id*. Díky této mapě při systémovém volání indentifikuje cílové vlákno na hostícím OS.

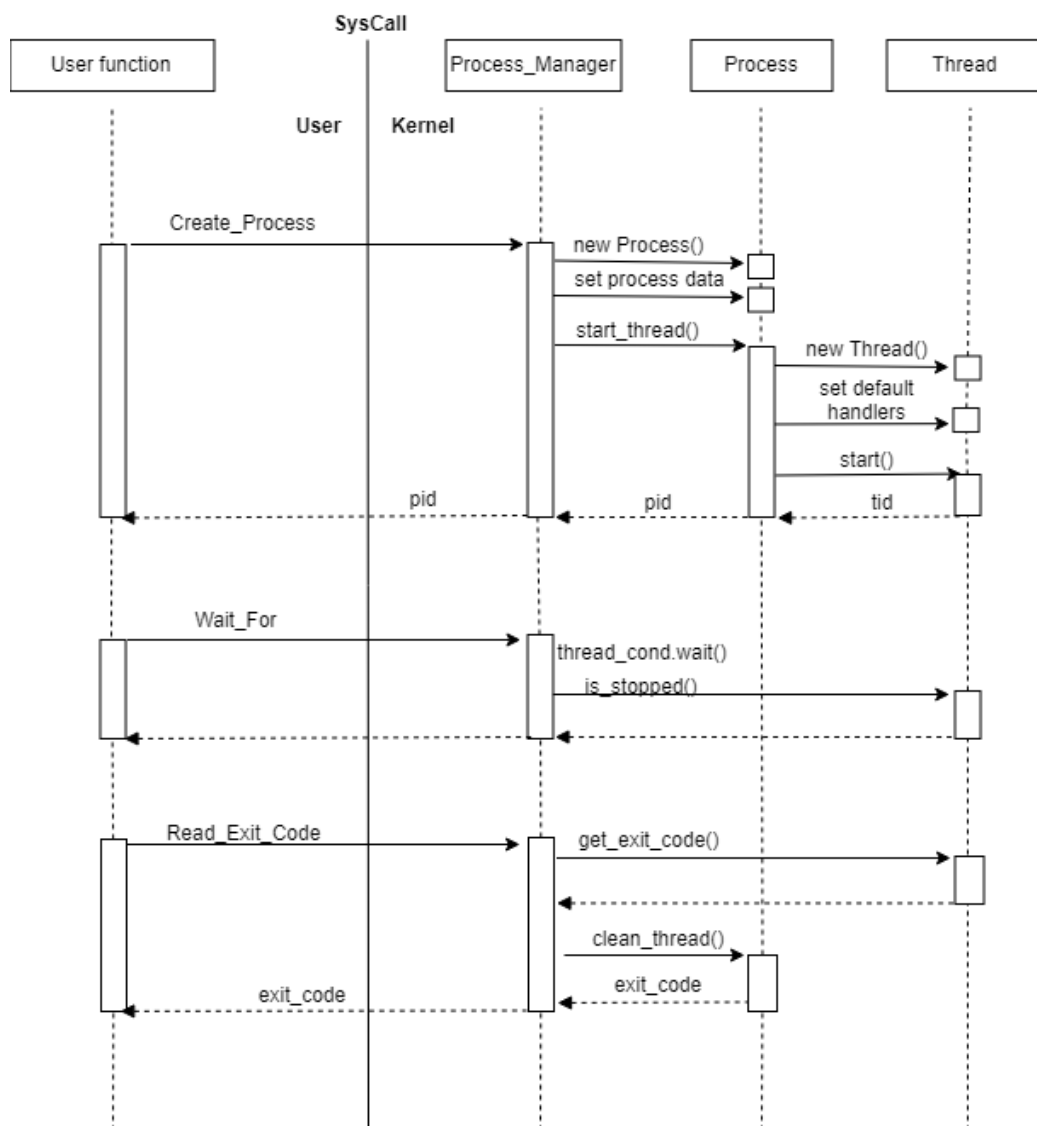
Všechny obsluhy systémových volání procesů probíhají kvůli nutnosti udržení interní konzistence dat atomicky.

Sekvenční diagram volání systémových služeb lze vidět na obrázku 2.3.

2.3.1 Synchronizace procesů

Synchronizace procesů je dle API zajištěna pomocí systémového volání `Wait_For`. Argumentem `Wait_For` je pole handleů procesů/vláken na které volající proces čeká. Jakmile první vlákno ukončí svůj běh, `Wait_For` se odblokuje a v návratové hodnotě předá handle ukončeného vlákna.

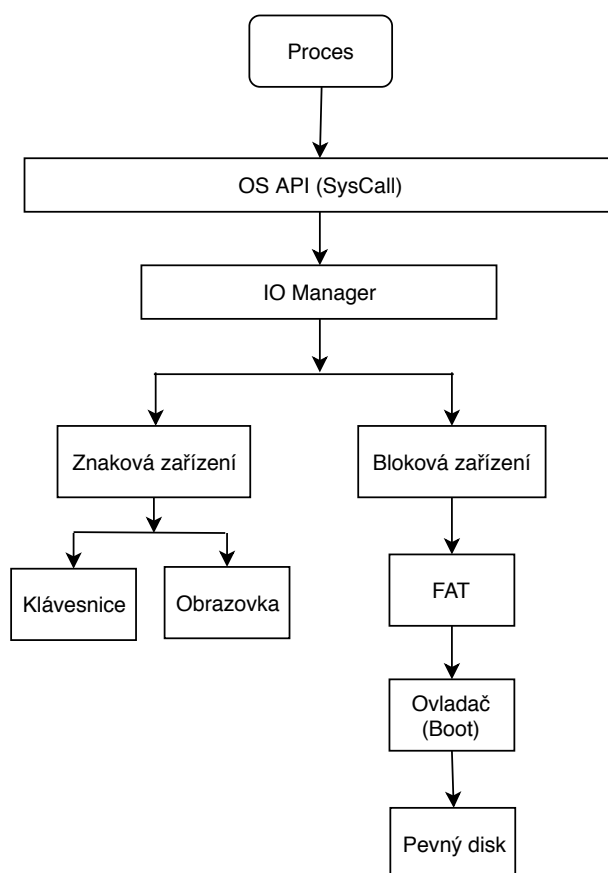
Funkcionalita `Wait_For` je implementována pomocí podmínkové proměnné. Jakmile se ukončí vlákno, změní svůj stav na `stopped` a signalizuje podmínkovou proměnnou. Funkce `Wait_For` se na podmínkové proměnné uspí. Jakmile ji ukončené vlákno probudí, zkontroluje, zda probuzené vlákno je na seznamu vláken, na které `Wait_For` čeká. Pokud ano, funkce se ukončí a vrátí handle ukončeného vlákna. V opačném případě se opět uspí a čeká na signalizování ukončení dalšího vlákna.



Obrázek 2.3: Sekvenční diagram volání systémových služeb OS.

3 IO

Na následujícím diagramu (obr. 3.1) je vidět struktura vstupních a výstupních zařízení. Jednotlivá zařízení poskytují stejné rozhraní pro práci s nimi. To usnadňuje jejich použití, a také následnou kontrolu oprávnění. IO Manager vytváří IO Handle, který může reprezentovat obrazovku, klávesnici, disk nebo pipe.



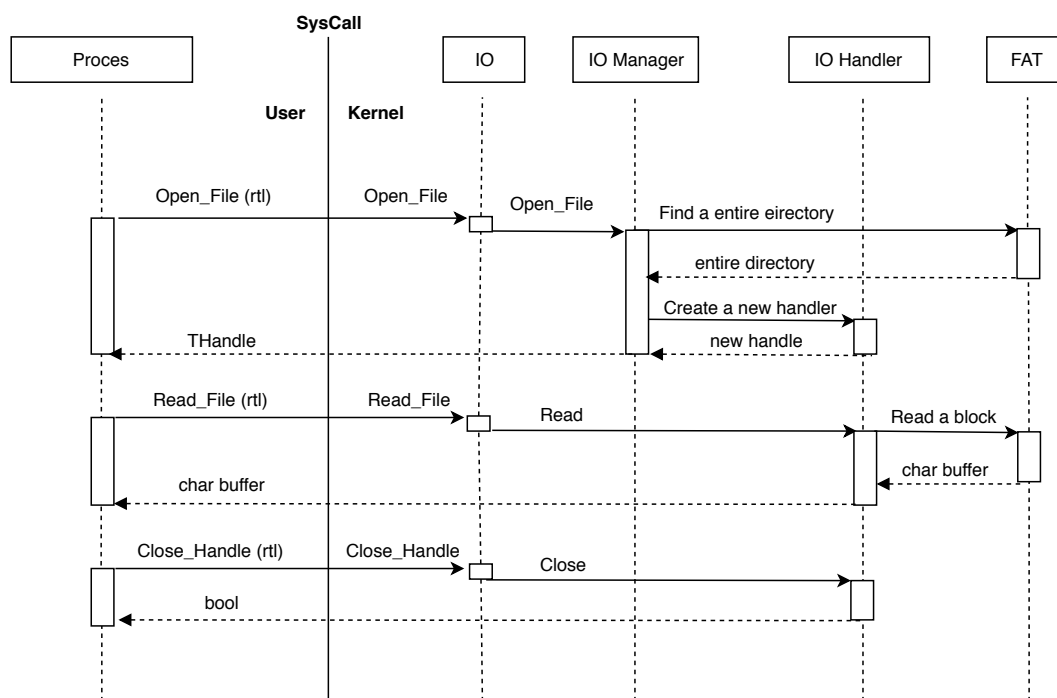
Obrázek 3.1: Architektura IO.

Na sekvenčním diagramu (obr. 3.2) je vidět, jak probhává volání vstupních a výstupních operací. V uživatelském procesu je vytvořen dotaz, kde jsou registry naplněny potřebnými daty. Zavoláním systémového volání je přepnuto do privilegovaného režimu jádra a dotaz je předán IO, který určí, jaká služba jádra byla zavolána. Pro práci s IO je nejdříve nutné vytvořit příslušný IO Handle, který obaluje přístup k jednotlivým operacím. Dotaz je

předán IO Manažeru, který vytvoří IO Handle podle zadaných parametrů. IO Handle je vrácen jako identifikátor THandle do uživatelského procesu.

Podobná situace nastává, pokud chceme nad daným identifikátorem zavolat služby poskytující RTL (např. čtení ze souboru). V jádru je načten příslušný obslužný IO Handle a zavolána obsluha tohoto volání. Například u čtení ze souboru jsou potřeba oblužné rutiny obalující funkčnost souborového systému. Výsledkem jsou načtená data, která jsou předána uživatelskému procesu.

Každý otevřený soubor je potřeba také korektně ukončit. Zavoláním služby pro uzavření IO Handle dojde k jeho smazání v jádru a do uživatelského procesu je vrácen stav operace.



Obrázek 3.2: Sekvenční diagram.

3.1 Souborový systém (FAT16)

Souborový systém byl zvolen FAT16 v mírně modifikované podobě. Modifikován je zejména proto, že nejsou využity všechny informace, které by měli být uloženy. Jedná se například o instrukce, které jsou uloženy na začátku disku nebo některé atributy ve vstupním bodu složky. Na disku jsou uloženy pouze informace, které jsou pro nás užitečné, případně jsou kom-

patibilní s API mezi jádrem a uživatelským prostorem. Souborový systém FAT využívá alokoční tabulku, ve kterých jsou uloženy čísla sektorů, kde se nachází části souboru.

3.1.1 Boot sector

Jedná se o první sektor, který se nachází na disku. Nachází se na disku vždy jeden a na obrázku (obr. 3.3) jsou znázorněny informace, které jsou relevantní pro správné fángování operačního systému.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	2E	3C	2E	¹ 4B	49	56	5F	46	41	54	32	² 00	02	01	01	00
010	³ 01	⁴ 40	00	A1	13	F8	⁵ 14	00	0A	00	01	00	00	00	00	00
020	00	00	00	00	00	00	00	00	00	00	00	56	4F	4C	55	4D
030	45	20	20	20	20	20	46	41	54	31	36	20	20	20	00	00
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Obrázek 3.3: Struktura Boot Block.

Význam důležitých uložených bytů:

1. název disku
2. velikost sektoru
3. počet FAT
4. počet root složek
5. počet sektorů pro uložení FAT

Význam nedůležitých uložených bytů:

- **0x00 – 0x02**: instrukce pro bootstrap program
- **0x0D**: počet bloků pro alokační jednotku
- **0x0E – 0x0F**: počet rezervovaných bloků
- **0x13 – 0x14**: počet bloků na disku

- **0x15**: mediální deskriptor
- **0x18 – 0x19**: počet bloků pro track
- **0x1A – 0x1B**: počet bloků pro heads
- **0x1C – 0x1F**: počet skrytých bloků
- **0x2B – 0x35**: název oddílu disku
- **0x36 – 0x3D**: identifikátor souborového systému

3.1.2 Entire Directory

Každý soubor uložený na disku musí obsahovat vstupní bod, který obsahuje metadata popisující tento soubor. Tyto informace jsou uloženy v Entire Directory. Jak už ze specifikace FAT vychází, za alokační tabulkou se nachází staticky alokované sektory pro Root Entire Directory. Na obrázku (obr. 3.4) je znázorněna struktura.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	¹ 41	00	00	00	00	00	00	00	² 54	58	54	³ 00	⁴ 00	00	00	00
010	00	00	00	00	00	00	⁵ 00	00	⁶ 00	00	⁷ 00	02	⁸ 32	00	00	00
020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Obrázek 3.4: Struktura Entire Directory.

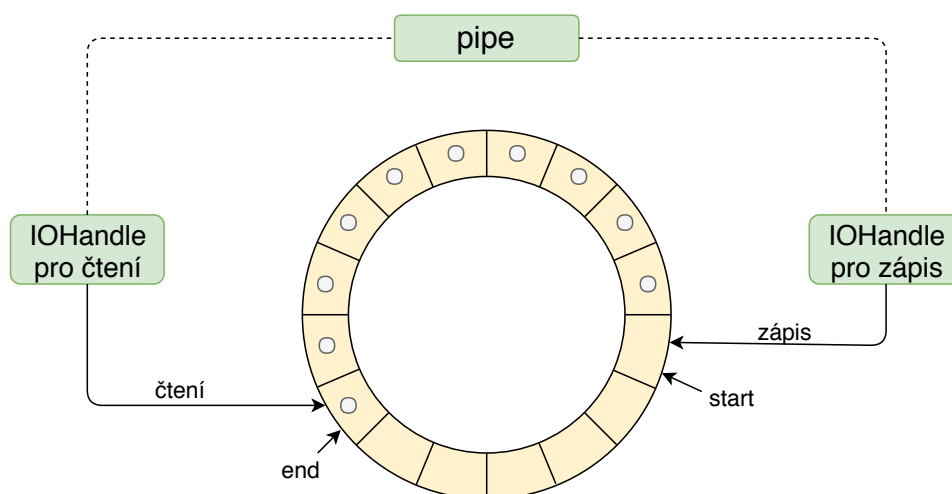
1. název souboru
2. formát souboru
3. atributy
4. rezervováno
5. čas vytvoření nebo aktualizace
6. datum vytvoření nebo aktualizace
7. první sektor
8. velikost (byte)

Poskytnuté API umožňuje přenos pouze atributy, názvu souboru a formát. Parametry datum a čas vytvoření jsou tedy nevyužity a jsou vždy nastavené na 0. Jádro je ovšem na to připravené a změnou API je možné tento údaj přenést. Taktéž nelze přenést velikost souboru, a tak je využita pouze pro kontrolu, zda je daný soubor/složka možné smazat či nikoliv.

3.2 Přesměrování a roury

Rouru představuje kruhový buffer umožňující dvěma procesům spolu komunikovat na základě modelu producent-konzument, který je v tomto případě realizovaný podmínkovou proměnou.

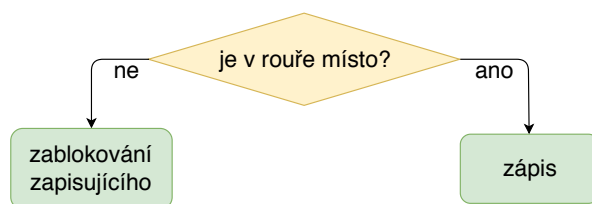
Přístup do kruhového bufferu je možný přes dva konce, kde první umožňuje pouze zápis a druhý pouze čtení. Tento princip je znázorněn na obrázku 3.5.



Obrázek 3.5: Roura

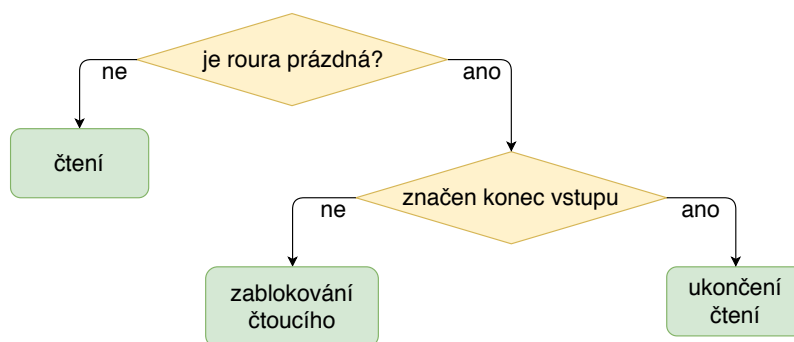
Při vytvoření má kruhový buffer pevně danou velikost a dva ukazatele `start` a `end` nastavené na 0. Ukazatel `start` odkazuje na index kruhového bufferu, kam může jeden proces zapisovat a `end` odkazuje na index, odkud lze číst. Do kruhového bufferu je tedy zapisováno po jednotlivých charech a stejně tak je z něj po jednotlivých charech také čteno. Dále rourě náleží flag `is_EOF`, který značí konec vstupu a jeho výchozí nastavení je `false`.

Když přijde rourě požadavek na zápis, je tento požadavek obsloužen pouze v případě, že je v kruhovém bufferu alespoň jeden index volný. V opačném případě je proces požadující zápis zablokovaný. Zápis do roury je naznačen na obrázku 3.6.



Obrázek 3.6: Zápis do roury

Obdobně funguje také obsluha požadavku pro čtení. Pokud je v rouře alespoň jedna položka, je tento požadavek obsloužen. Pokud je roura prázdná, je proces zablokován. Výjimku zde tvoří případ, kdy je flag `is_EOF` nastaven na hodnotu `true`. V tomto případě může proces pokračovat i přesto, že se v rouře nenachází žádné položky. Flag `is_EOF` s hodnotou `true` je ovšem v případě prázdného bufferu zachycen ještě před samotným čtením a čtoucímu procesu je navracena hodnota -1, značící konec vstupu. Tento algoritmus je zachycen na diagramu 3.7.



Obrázek 3.7: Čtení z roury

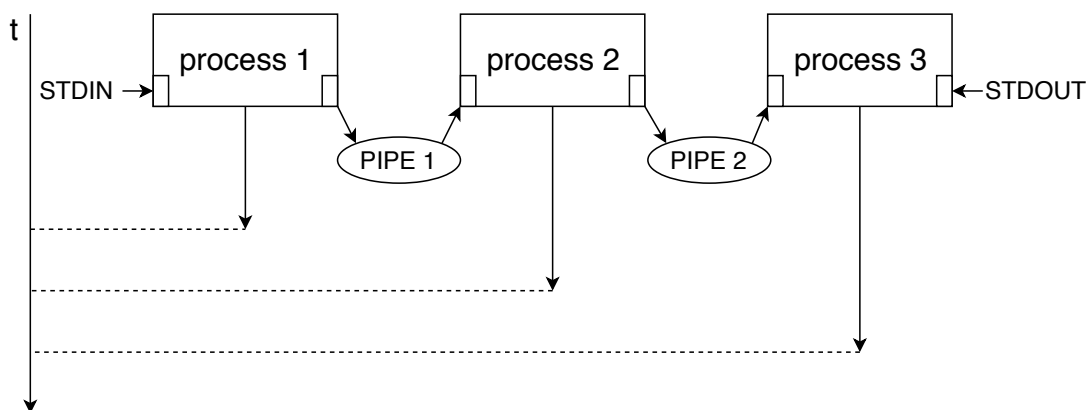
4 Shell

Shell v podobě příkazového řádku umožňuje uživateli interakci s programem. Vstup od uživatele čte do té doby, než přijde znak Ctrl+Z / EOF, případně dokud uživatel nezavolá funkci `exit` pro ukončení aktuálního shellu nebo funkci `shutdown` pro ukončení aktuální instance programu.

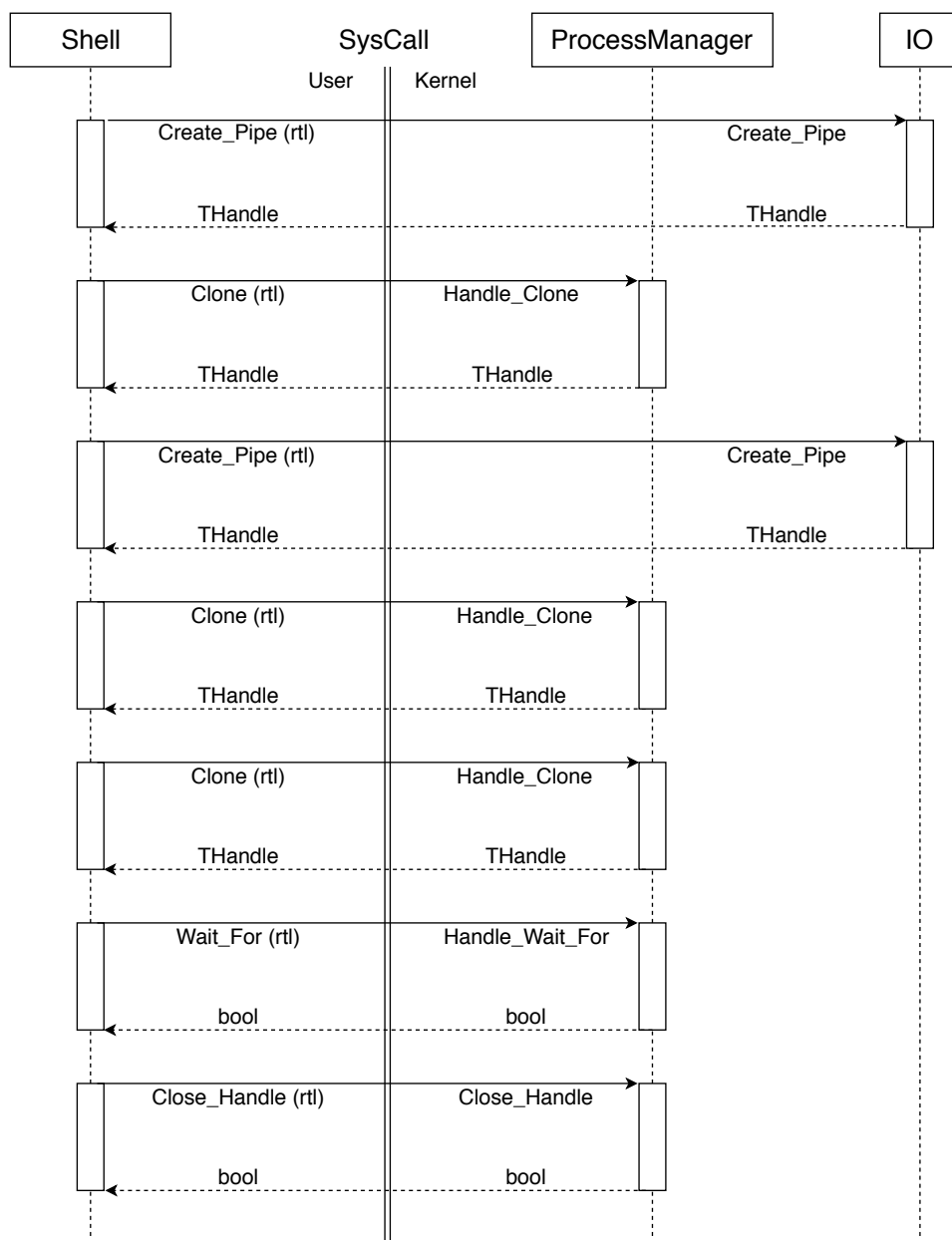
Vstup je poté parsován, také s ohledem na funkčnost `rour` a přesměrování. Díky tomu, že je stream dostatečně abstrahovaný, proces s ním v rámci vstupu i výstupu pracuje stejně, ať už se jedná o vstup/výstup ze souboru nebo jiné funkce. Platí, že vstup první funkce je vždy buď `stdin` nebo ze souboru a výstup poslední funkce je buď na `stdout` nebo je přesměrován do souboru.

Roury slouží jako prostředník dvou funkcí. První funkce má jako výstup odkaz na `IOHandle` pro zápis do roury a druhá funkce má jako vstup odkaz na `IOHandle` pro čtení z roury. Funkce jsou spuštěny paralelně. Po ukončení daných procesů jsou již nepotřebné handlers pozavírány.

Příklad použití `rour` mezi jednotlivými procesy je znázorněn na obrázku 4.1. Sekvenční diagram pro stejný příklad ukazující jednotlivá systémová volání je na obrázku 4.2.



Obrázek 4.1: Provedení příkazu `Program1 | Program2 | Program3`



Obrázek 4.2: Provedení příkazu Program1 | Program2 | Program3

5 Uživatelské funkce

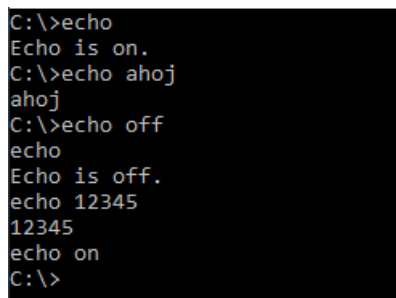
5.1 echo

Příkaz `echo` umožňuje několik způsobů použití. Lze pomocí něj vypsat zadaný vstup nebo zapnout/vypnout zobrazování aktuálního pracovního adresáře.

Výchozí nastavení zobrazování aktuálního pracovního adresáře je zapnuté. Vypnout jej lze zadáním příkazu `echo off` a opět zapnout příkazem `echo on`. Aktuální nastavení lze zjistit zadáním příkazu `echo` bez parametrů. V případě, že je zobrazování aktuálního pracovního adresáře zapnuto, je vypsána informace „Echo is on.“, v opačném případě je vypsáno „Echo is off.“.

Pokud je příkaz `echo` zadán s jakýmkoliv jinými parametry, případně i kombinací parametrů `on` a `off` s jinými nebo se sebou navzájem, jsou tyto parametry pouze vypsány.

Ukázka použití je na obrázku 5.1.



```
C:\>echo
Echo is on.
C:\>echo ahoj
ahoj
C:\>echo off
echo
Echo is off.
echo 12345
12345
echo on
C:\>
```

Obrázek 5.1: Příkaz `echo`

5.2 cd

Příkaz `cd` má za úkol změnu aktuálního pracovního adresáře. Má jeden parametr, který značí buď relativní, nebo absolutní cestu k novému pracovnímu adresáři. Pokud se uživatel pokusí přistoupit do neexistujícího adresáře, je mu navracena chyba „File not found.“ Ukázka změny aktuálního pracovního adresáře je na obrázku 5.2.

```
C:\>cd a
C:\a>cd b
C:\a\b>cd ..
C:\a\>
```

Obrázek 5.2: Příkaz cd

5.3 dir

Příkaz `dir` vypisuje aktuální obsah adresáře. Parametrem může být cesta, na které se má obsah adresáře vypsat. Výstupem je seznam souborů a složek, které mají následující strukturu - název souboru, označení FILE/DIR, přístupová práva (R nebo R/W). Příklad správného použití je naznačen na obrázku 5.3.

```
C:\>dir
foobar.txt      <FILE>  R
loremips.txt    <FILE>  R/W
lorem2.txt      <FILE>  R/W
a               <DIR>   R/W
C:\>
```

Obrázek 5.3: Příkaz dir

5.4 md

Příkaz `md` slouží k vytváření nového adresáře. Má jediný parametr, a to cestu nového adresáře. Ta může být zadána jak v relativní, tak v absolutní podobě. V případě, že byl pokus o vytvoření úspěšný, nic se nevypíše. Uživatel je tedy informován pouze o případné chybě.

```
C:\>dir
C:\>md nový

C:\>dir
nový      <DIR>  R/W
C:\>
```

Obrázek 5.4: Příkaz md

5.5 rd

Příkaz `rd` maže zadaný adresář. Parametrem je název mazané složky, která může být zadána jak relativně, tak absolutně. Příklad použití je na obrázku 5.5.

```
C:\>dir
test                <DIR>    R/W
C:\>rd test

C:\>dir
C:\>
```

Obrázek 5.5: Příkaz `rd`

5.6 type

Příkaz `type` má dva způsoby použití. Při spuštění bez parametru čte se `stdin` a následně tento vstup vypíše. Jako parametr je možné zadat název souboru a pokud tento soubor existuje, je jeho obsah vypsán na `stdout`.

Příklad použití `type` bez parametru a s parametrem je naznačen na obrázku 5.6.

```
C:\>type
Hello
World !
Hello
World !
C:\>type loremips.txt
1Lorem ipsum dor sit amet, consectetur adipiscing elit.
bulum neque, non pretium dui lacus et lacus. Mauris ege
erat blandit, sodales nisl eget, dignissim lorem. Fusc
at tristique, lacus quam placerat orci, in consequat ip
lobortis hendrerit turpis dui.23Lorem ipsum dor sit am
ur imperdiet, dolor mauris vestibulum neque, non pretiu
is, accumsan turpis. Vivamus nec erat blandit, sodales
uam. Nunc semper, eros eu placerat tristique, lacus qua
```

Obrázek 5.6: Příkaz `type`

5.7 find /v /c""

Příkaz `find /v /c""` je obdobou unixového `wc`. Čte ze standardního vstupu a následně spočte počet řádek vstupu.

Příklad použití příkazu `find /v /c""` je na obrázku 5.7.

```
C:\>find /c /v ""  
prvni radka  
druha radka  
treti radka  
3  
C:\>
```

Obrázek 5.7: Příkaz `find /v /c ""`

5.8 sort

Příkaz `sort` čte ze standardního vstupu a po jeho ukončení seřadí všechny řádky vzestupně na základě jejich binární podoby. Primární použití je zejména při přesměrování výstupu procesu do roury. Ukázka použití je na obrázku 5.8.

```
C:\>sort  
prvni  
druha  
treti  
druha  
prvni  
treti  
C:\>
```

Obrázek 5.8: Příkaz `sort`

5.9 tasklist

Příkaz `tasklist` je obdobou unixového `ps` a slouží k vypsání všech běžících procesů. Každý proces zde má uvedenou informaci o názvu vykonávané uživatelské funkce, svém stavu (`prepared`, `running`, `stopped`) a id (PID – process id).

Ukázka použití příkazu `tasklist` je znázorněn na obrázku 5.9.

```
C:\>tasklist  
ps      running 1  
shell   running 0  
C:\>
```

Obrázek 5.9: Příkaz `tasklist`

5.10 shutdown

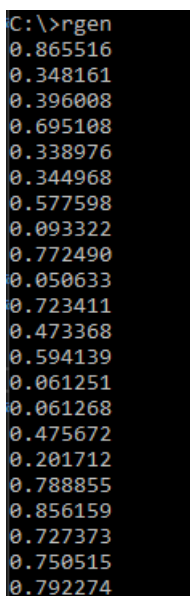
Příkaz `shutdown` má jediný úkol, a to ukončit aktuální instanci programu.

5.11 rgen

Příkaz `rgen` generuje pomocí Mersenne Twisteru náhodná čísla v plovoucí čárce tak dlouho, dokud mu nepřijde znak Ctrl+Z, EOF nebo EOT.

Tato funkce je řešena dvouvláknově. Hlavní vlákno procesu generuje pseudonáhodná čísla a další vlákno kontroluje, zda na vstup nepřišel znak představující konec vstupu. Obě tato vlákna mají zároveň zaregistrovaný `Sigterm Handler`, aby mohla být při `shutdownu` ukončena.

Ukázka generování čísel v plovoucí čárce pomocí příkazu `rgen` je na obrázku 5.10.



```
C:\>rgen
0.865516
0.348161
0.396008
0.695108
0.338976
0.344968
0.577598
0.093322
0.772490
0.050633
0.723411
0.473368
0.594139
0.061251
0.061268
0.475672
0.201712
0.788855
0.856159
0.727373
0.750515
0.792274
```

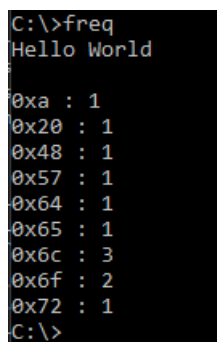
Obrázek 5.10: Příkaz `rgen`

5.12 freq

Příkaz `freq` čte ze standardního vstupu a po jeho ukončení sestaví frekvenční tabulku všech zadaných bytů. Všechny její položky, jenž mají nenulovou frekvenci, jsou poté vypsány ve formátu `0x%hhx : %d`, přičemž první

parametr je zadaný byte a druhý parametr je jeho frekvence v zadaném vstupu.

Příklad použití `freq` je na obrázku 5.11.



```
C:\>freq
Hello World

0xa : 1
0x20 : 1
0x48 : 1
0x57 : 1
0x64 : 1
0x65 : 1
0x6c : 3
0x6f : 2
0x72 : 1
C:\>
```

Obrázek 5.11: Příkaz `freq`

6 Závěr

K semestrální práci byla k dispozici kostra, která implementovala simulaci bootu a ukázkou programu Shell. Při implementaci všech uživatelských programů jsme využili API operačního systému. Jádru obsahuje implementaci všech potřebných částí tohoto API.

Provedli jsme několik testů přes dodaný program *checker.exe*, kde jsme změřili průměrnou dobu běhu. Průměrný čas byl 533.7 ms. Test proběhl na školním počítači s konfigurací Windows 10, Intel Xeon E3-1246 v3 3.50 GHz, RAM 32 GB. Při testování jsme se také zaměřili na využití paměti a zkusili pustit cca tisíc příkazů. Paměť se držela ve stabilní poloze a nealokovalo se stále více a více paměti. Těmito testy jsme usoudili, že vytvořené řešení je kvalitní pro úroveň semestrální práce tohoto předmětu.