Katedra informatiky, FEI VŠB-TUO, Petr Olivka.

Tento text je neautorizovaný a nerecenzovaný překlad doporučené literatury: „Andrew S. Tanenbaum, Operating Systems: Design and Implementation“, a je určen jen pro studijní účely.

1 Operační systém

1.1 Co to je operační systém

Většina uživatelů počítačů má určité zkušenosti s operačním systémem, ale je těžké přesně vysvětlit, co to je operační systém. Část problému je v tom, že operační systém zajišťuje dvě základní, ne zcela související funkce a velmi záleží na tom, koho zajímají. Tak se na ně podívejme.

1.1.1 Operační systém jako stroj s rozšířenými možnostmi

Jak je známo, počítačová architektura (sada instrukcí, organizace paměti, vstupy a výstupy, sběrnice atd.) většiny počítačů je velmi primitivní a velmi nepříjemně se programuje, zejména vstupy a výstupy (V/V). Abychom si to přiblížili, podívejme se trochu blíže, jak probíhají V/V operace z diskety pomocí NEC PD765 řadiče, nebo ekvivalentu, používaného ve většině osobních počítačů. PD765 má 16 příkazů, každý specifikovaný 1 až 9 byty, zasílanými do registru řadiče. Tyto příkazy jsou pro čtení a zápis dat, pohyb čtecí hlavy, formátování stopy, inicializaci, identifikaci, reset a kalibraci řadiče i mechaniky.

Nejzákladnější příkazy jsou READ a WRITE, každý vyžaduje 13 parametrů zabalených do 9 bytů. Tyto parametry specifikují takové údaje, jako adresa diskového bloku pro čtení, počet sektorů na stopu, záznamový režim použitý na médiu, oddělovací mezeru mezi sektory, nebo jak manipulovat se značkou mazané datové adresy. Pokud tomuto názvosloví nerozumíte, není třeba se bát, to je přesně ten bod jen pro zasvěcené. Když je operace kompletní, řadič vrátí status 23 a chybové pole zabalené do 7 bytů. A aby toho ještě nebylo dost, programátor floppy řadiče si musí být ještě vědom, zda je motor mechaniky zapnutý, či nikoliv. Pokud je vypnutý, musí se s velikým předstihem před čtením a zápisem dat zapnout. A motor se pak nesmí nechat dlouho zbytečně zapnutý, aby se nepřehřál. Tady musí programátor udělat kompromis mezi dlouhou dobou rozběhu mechaniky a jejím přehříváním.

Bez toho, že bychom se zabývali skutečnými detaily, musí být zcela jasné, že průměrný programátor se pravděpodobně nebude chtít seznamovat s detaily komplikovaného programování diskety a disku. To, co programátor potřebuje, je jen jednoduchá, vysokoúrovňová abstrakce na manipulaci s ní. V případě disku je typickou abstrakcí, že disk obsahuje skupinu pojmenovaných souborů. Každý soubor může být otevřen pro čtení i zápis, následně

se data čtou či zapisují a nakonec se uzavře. Detaily, jako zda použít modifikovanou frekvenční modulaci pro záznam a jaký je aktuální stav motoru, by se na abstraktní úrovni objevit neměly.

Program, který ukrývá realitu hardware před programátorem a prezentuje „příjemný“ a jednoduchý pohled na pojmenované soubory pro čtení i zápis, to je to očekávané, tedy operační systém. Právě tak, jako odděluje operační systém programátora od hardware disku a prezentuje se jako jednoduché, souborově orientované rozhraní, stejně tak ukrývá celou řadu dalších nepříjemných funkcí, jako přerušení, časovače, správu paměti a další nízkoúrovňové funkce a vlastnosti. V každém případě, abstrakce nabízená operačním systémem je jednodušší a snadnější pro použití, než základní hardware.

Z tohoto pohledu je funkcí operačního systému prezentovat uživateli **stroj s rozšířenými možnostmi**, nebo taky jako **virtuální stroj**, který je pro programování jednodušší. Jak toho operační systém dosáhne, je povídání na dlouho a budeme to zde dále probírat podrobněji.

1.1.2 Operační systém jako správce prostředků

Koncepce operačního systému primárně nabízejícího uživateli přívětivé rozhraní, je pohled z hora–dolů. Alternativně, pohledem zdola–nahoru, operační systém drží a spravuje všechny části komplexního systému. Moderní počítače se skládají z procesoru, paměti, časovačů, disků, myši, síťového rozhraní, tiskárny a další široké škály zařízení. V alternativním pohledu, úkolem operačního systému je zajistit řádný a řízený přístup k procesoru, paměti, V/V zařízením mezi různými program, o ně se ucházejícími.

Představme si, co se stane, pokud se tři spuštěné programy na jednom počítači pokusí současně tisknout na stejné tiskárně. Několik prvních řádků bude z programu 1, pak budou následovat řádky z programu 2 a několik z programu 3, a tak stále dokola. Výsledkem bude chaos. Operační systém může přinést pořádek do potencionálního chaosu ukládáním všech výstupů na tiskárnu na disk. Až jeden program skončí, může operační systém zkopírovat jeho výstup z diskového souboru, kde byl uložen, na tiskárnu, zatím co souběžně mohou ostatní programy pokračovat v generování dalších výstupů díky tomu, že jejich výstup není reálně směrován na tiskárnu.

Když je na počítači více uživatelů, je potřeba spravovat a chránit paměť, V/V zařízení a další zdroje daleko více, aby si uživatelé nepřekáželi a vzájemně se neobtěžovali. Musíme dodat, že uživatelé nepotřebují sdílet jen hardware, ale i informace v souborech a databázích. Ve zkratce, tento pohled na operační systém má jako hlavní úkol sledovat, kdo používá jaké zdroje, vyhovět žádostem o zdroje, řídit jejich používání a urovnávat konflikty mezi požadavky od různým programů i uživatelů.

1.2 Historie operačních systémů

Operační systémy se vyvíjí mnoho let. Podívejme se v následujícím textu stručně na přehled vývoje. Tím, že je operační systém historicky blízce svázán s architekturou počítače na kterém běží, tak bude tento pohled odpovídat generacím počítačů, aby bylo zřejmé, jaké byly jejich operační systémy. Toto spojení generací počítačů a operačních systémů je velmi nepřesné, nabízí však alespoň nějakou formu rozdělení, zatím co jiná spojení žádnou.

První opravdový digitální počítač byl navržen anglickým matematikem Charlesem Babbagem (1792-1871). Ačkoliv Babbage věnoval vývoji „analytického stroje“ většinu života i majetku, nikdy nevytvořil spolehlivě fungující stroj, protože byl čistě mechanický a dobové technologie neumožnily vyrobit kolečka, řazení a ozubení s potřebnou přesností. Asi není třeba říkat, že tento stroj neměl operační systém.

Co je na této historii zajímavé, že Babbage potřeboval pro svůj stroj programy a pronajal si mladou ženu jménem Ada Lovelance, což byla dcera známého britského básníka Lorda Byrona, jako prvního programátora na světě. Programovací jazyk Ada je po ní pojmenován.

1.2.1 První generace (1945-1955) - elektronky a zásuvné desky

Od neúspěšného pokusu Babbage, až do druhé světové války, vývoj digitálních počítačů nepokročil. Až v polovině 40. let Howard Aiken na Harvardu, John von Neumann v Princetonu, J.Presper Ecker, William Mauchley na univerzitě v Pensylvánii a Konrad Zuse v Německu, z řady mnoha dalších, uspěli se stavbou výpočetního stroje pomocí elektronek. Tyto stroje byly obrovské, zabíraly několik místností se svými desítkami tisíc elektronek, a byly pomalejší, než nejlevnější z dnešních osobních počítačů. V této rané době jen skupina lidí navrhovala, stavěla, programovala, obsluhovala a spravovala všechny stroje. Veškeré programování bylo čistě jen ve strojovém kódu, nejčastěji přímo zapojením na zásuvných deskách pro řízení základních funkcí. Programovací jazyk byl pojem neznámý, dokonce nebyl ani assembler, tedy jazyk symbolických instrukcí. Něco jako operační systém byla v té době věc neslýchaná. Obvyklý způsob obsluhy byl pro programátora takový, že si na nástěnce v rozvrhu zarezervoval určitý čas, kdy může jít do místnosti s počítačem a zasunout svou zásuvnou desku do počítače a strávit několik následujících hodin v dobré naději, že těch 20000 elektronek vydrží žhavit po dobu výpočtu. Prakticky všechny výpočty byly přímé numerické výpočty, jako vyplňování tabulek funkcí sinus a cosinus.

Až na počátku 50. let se tento postup poněkud vylepšil s vynálezem děrných štítků. Nyní bylo možné napsat program na štítky a načíst jej, místo použití zásuvných desek. Jinak se nic nezměnilo.

1.2.2 Druhá generace (1955-65) - tranzistory a dávkové systémy

Vynález tranzistoru v polovině 50. let radikálně změnil možnosti. Počítače se staly dostatečně spolehlivé, aby mohly být vyráběny a prodávány zákazníkům s tím, že budou pracovat dostatečně dlouho, aby odvedly dobrou práci. Poprvé se začaly oddělovat profese návrháře, konstruktéra, obsluhy, programátora a údržby.

Aby počítače fungovaly, byly zavřeny v klimatizovaných místnostech s profesionální obsluhou. Jen velké organizace, vládní agentury nebo univerzity si mohly dovolit investovat několik miliónů dolarů do jejich pořízení. Před spuštěním **dávky**, tj. programu, nebo skupiny programů, musel programátor napsat program na papír ve Fortranu nebo assembleru a pak vyděrovat na štítky. Pak vzal krabici se štítky, zašel do vstupní místnosti a předal ji obsluze.

Když počítač ukončil jakoukoliv dávku, obsluha zašla k tiskárně a vyzvedla výstup a zanesla jej do výstupní místnosti. Tam si je programátoři museli potom vyzvednout, rozdělit a zkompletovat. Pak si ještě vyzvedli krabici s děrnými štítky a mohli si jít číst. Pokud bylo potřeba kompilátor FORTRANu, operátor jej přinesl ze skříně se soubory a načetl jej. Poměrně hodně procesorového času bylo ztraceno během pochůzek po počítačovém sále.

Vzhledem k vysoké ceně zařízení není překvapení, že lidé začali rychle hledat cestu, jak minimalizovat časové ztráty. Jako obecné řešení je **dávkový systém**. Princip spočívá v tom, že se všechny dávky seřadí ve vstupní místnosti a pomocí malého počítače se nahrají na magnetickou pásku. Ta se přenese na samotný výpočetní počítač, kde jsou obvykle tři páskové mechaniky: vstupní, systémová a výstupní.

Až jsou všechny dávky nahrány na pásku, což může být zhruba hodina, páska se převine a zanese k hlavnímu počítači, kde se vloží jako vstupní. Obsluha spustí speciální program, jakýsi předchůdce dnešního operačního systému, který čte jednotlivé dávky z pásky a spouští je. Výstup je zapisován na pásku, místo přímého tisku. Až je jedna dávka ukončena, načítá se z pásky další a spouští. Když jsou všechny dávky zpracovány, obsluha vymění vstupní a výstupní pásku a výstupy zanese k **off-line** tiskárně s čtečkou pásky.

Struktura typické dávky je asi následující:

- `$JOB Time,ID-Number,Name`
- `$FORTRAN`
- `Fortran program ...`
- `$LOAD`
- `$RUN`

- `Data for programm ...`
- `$END`

Dávka začíná kartou $JOB, specifikující maximální množství času potřebné pro vykonání dávky v minutách, identifikaci účtu a jméno programátora. Pak přichází karta $FORTRAN, instruující operační systém, aby spustil překladač FORTRANu z pásky. Ten očekává zdrojový text programu. Další štítek $LOAD přikazuje natáhnout do paměti přeložený program. Karta $RUN říká operačnímu systému, aby se program spustil s daty, které ho následují. Poslední karta $END označuje konec dávky. Tyto primitivní karty jsou předchůdcem moderních dávkových souborů a interpretů příkazových řádků.

Velké počítače druhé generace byly použity převážně pro vědecké a inženýrské výpočty, jako je řečení parciálních diferenciálních rovnic. Byly nejčastěji programovány ve FORTRANu a assembleru.

1.2.3 Třetí generace (1965-80) - int. obvody a multiprogramování

Na začátku 60. let měla většina výrobců počítačů dvě odlišné, zcela nekompatibilní produktové řady. Na jedné straně binárně orientované rozsáhlé počítače pro vědecké výpočty. Na straně druhé znakově orientované komerční počítače pro třídění a tisky v bankovním a pojišťovacím sektoru.

Vývoj a servis dvou kompletně odlišných produktových řad byl pro výrobce drahou záležitostí. K tomu mnoho nových uživatelů počítačů požadovalo malé počítače, později rozšiřitelné a požadovali velké počítače, na kterých poběží jejich stávající programy, ale rychleji.

IBM se pokusila tento problém řešit jednou řadou počítačů System/360. Série 360 byla programově kompatibilní řada počítačů od nejmenších až po ty nejvýkonnější. Počítače se lišily jen svou cenou a výkonem (maximální množství paměti, rychlost procesoru, povolený počet V/V zařízení, atd.). I když všechny počítače měly stejnou architekturu i množinu instrukcí, přenositelnost programu napsaného pro jeden počítač na ostatní, byla jen teoretická. Dále byla řada 360 navržena pro vědecké i komerční účely. Tak mohl výrobce jednou řadou počítačů uspokojit požadavky všech zákazníků. V následujících letech přišla firma IBM s následníky řady 360, modernějšími řadami 370, 4300, 3080 a 3090.

Řada 360 byla první hlavní linií používající integrované obvody s nízkou integrací. To přineslo významnou výhodu v poměru cena/výkon oproti druhé generaci, sestavené čistě jen z tranzistorů. To mělo okamžitý úspěch a myšlenka rodiny kompatibilních počítačů inspirovala i ostatní významné výrobce té doby. Potomci těchto počítačů se stále používají v některých počítačových centrech, ale jejich užívání rychle klesá.

Největší výhoda jediné řady počítačů je paralelně i její nevýhodou. Hlavní záměr byl, aby veškeré programy, včetně operačního systému pracovaly na

všech modelech. Aby fungovaly na malých systémech, kde se kopírují děrné štítky na pásky i na velkých systémech pro předpověď počasí a pro rozsáhlé výpočty. A taky aby byly dobré pro systémy s několika periférierniemi i pro systémy s velkým počtem periférií. A neposlední řadě, aby počítač pracoval v komerční sféře a pro vědecké účely. Zkrátka, aby byl počítač vyhovující pro všechny různorodé požadavky uživatelů.

Nenašlo se ale řešení, že by IBM mohla napsat program splňující všechny rozporuplné požadavky. Výsledkem byl rozsáhlý a velmi neobyčejný komplexní operační systém. Sestával z miliónů řádku v assembleru napsaného tisíci programátorů a obsahující tisíce a tisíce chyb. Každá nová verze opravila některé chyby a přinesla chyby nové, a tak počet chyb v systému byl průběžně konstantní.

Jeden z návrhářů OS/360, Fred Brook, v této souvislosti napsal veselou i kritickou knihu o jeho zkušenostech s OS/360 „The Mythical Man-Month". Sice zde není dostatek prostoru knihu probírat a dělat závěry, ale určitě stačí říct, že na obale je stádo prehistorických zvířat, která se nemohou pohnout z místa.

Navzdory obrovskému množství problémů, OS/360 a podobné operační systémy třetí generace produkované dalšími výrobci, uspokojili v té době většinu jejich zákazníků poměrně slušně. Také zpopularizovali několik nových technik chybějících v druhé generaci operačních systémů. Pravděpodobně nejdůležitější z nich bylo **multiprogramování**. Když na počítači aktuální dávka čekala na data z pásky nebo jinou V/V operaci, procesor zůstal ve stavu čekání na V/V požadavek. U procesorů (CPU) určených pro rozsáhlé vědecké výpočty jsou V/V operace ojedinělé a ztráta času není nijak významná. U komerčního zpracování dat mohou být V/V operace často 80 až 90% z celkového času a je proto třeba něco udělat, aby CPU nečekal příliš dlouho.

Řešení, které se vyvinulo, spočívalo v rozdělení paměti na několik bloků (partitions) s různými dávkami v každém z nich. Zatím co jedna dávka čekala na ukončení V/V operací, jiná dávka mohla využívat CPU. Pokud mohla paměť pojmout dostatečné množství dávek současně, mohlo být využití CPU až 100%. Mít ale v paměti více dávek vyžadovalo i mít jiný hardware na ochranu jednotlivých dávek mezi sebou proti sledování i chybám ostatních. Počítače třetí generace na to vybaveny byly.

Další novou vlastností počítačů třetí generace byla schopnost číst dávky z děrných štítků na disky, jakmile byly přineseny na počítačový sál. Pak, jakmile běžící dávka skončila, operační systém mohl natáhnout z disku do uvolněného bloku paměti další dávku a spustit ji. Této technice říkáme **spooling** (Simultaneous Peripheral Operation On Line) a byl také použit pro výstup. Se spoolingem nebyly třeba počítače pro čtení štítků na pásky a zmizela částečně nutnost přenášet pásky.

Ačkoliv třetí generace operačních systémů byla vyhovující pro velké vědecké výpočty i pro zpracování rozsáhlých komerčních dat, stále to byl od

základu dávkový systém. Mnoho programátorů znalých první generaci počítačů, kdy měli počítač na několik hodin jen pro sebe, tehdy mohlo ladit své programy rychleji. S třetí generace systémů byl čas mezi přijetím dávky a získáním výsledků i několik hodin a chybějící středník mohl způsobit chybu kompilátoru a programátor ztratil půl dne.

Tento požadavek na rychlejší odezvu vydláždil cestu ke **sdílení času** (timesharing) ve kterém každý uživatel má on-line terminál. V systému se sdíleným časem s 20 uživateli, kde 17 z nich popíjí kávu, povídá si nebo přemýšlí, může být CPU alokováno pro 3 dávky, které potřebují provést. Programátoři ladící své programy obvykle užívají krátké příkazy raději než dlouhé a tak může počítač rychle a interaktivně zpracovat požadavky mnoha uživatelům a možná také pracuje na velkých dávkách na pozadí, když CPU je v čekacím stavu. Ačkoliv první systém se skutečným sdílením času byl vyvinut v roce 1962, sdílení času se nestalo příliš oblíbené, dokud se potřebné ochrany hardware nestaly dostatečně rozšířené v rámci třetí generace.

Jeden z počítačových expertů v Bellových laboratořích, který pracoval na systému MULTICS (MULTiplexed Information and Computing Service), Ken Thompson, našel jeden malý nepoužívaný počítač PDP-7 a napsal zjednodušenou jednouživatelskou verzi systému MULTICS. Tato práce se časem změnila v operační systém UNIX, který se stal populární v akademické sféře, ve vládních agenturách a mnoha organizacích.

Historie UNIXu je popsána v mnoha pramenech. Řekněme jen, že díky dostupnosti zdrojového kódu, mnoho organizací začalo vyvíjet své vlastní (nekompatibilní) verze, což vedlo k chaosu. Aby byla možnost napsat program, který by mohl běžet na jakémkoliv UNIXu, IEEE vydalo standard pro UNIX, známý jako POSIX, který je UNIXy v dnešní době podporován. POSIX definuje minimální množinu systémových volání, které musí systém UNIX implementovat. Standardu POSIX dnes odpovídají i další operační systémy.

1.2.4 Čtvrtá generace (1980-nyní) - personální počítače

S rozvojem číslicových obvodů LSI (Large Scale Integration), kdy obvod obsahuje tisíce tranzistorů na cm^2 křemíku, přišel čas personálních počítačů. Podle architektury je personální počítač stejný jako mikropočítač třídy PDP-11, ale zásadně se liší v ceně. Zatím co mikropočítač je vhodný pro oddělení podniku či katedry, aby měly vlastní počítač, implementace mikroprocesorového čipu je vhodná pro jednotlivce, který potřebuje mít svůj vlastní počítač. Nejvýkonnější personální počítače používané v obchodě, na univerzitách nebo ve státních organizacích obvykle nazýváme **pracovní stanicí** (workstation), ale doopravdy to jsou velké personální počítače. Obvykle jsou propojeny v síti.

Velké rozšíření a dobrá dosažitelnost výpočetního výkonu, zejména velmi interaktivní výpočetní výkon obvykle s výbornou grafikou, vede k rozvoji

průmyslu vyvíjejícího programy pro personální počítače. Většina tohoto software je uživatelsky přátelská, aby mohla být určena pro uživatele, kteří neví nic o počítačích i nikdy nebudou mít snahu se to učit. To byla hlavní změna proti OS/360, jehož dávkový řídící jazyk byl tajemstvím, dokud o něm nevyšla kniha v roce 1970.

Dva operační systémy dominovaly na začátku ve sféře personálních počítačů: Microsoft MS-DOS a UNIX. MS-DOS byl převážně použit na počítačích IBM PC a dalších, založených na procesorech Intel 8088 a jeho následnících, 80286, 80386 a 80486. Později Pentium a Pentium Pro. Ačkoliv první verze MS-DOSu byla relativně primitivní, následné verze obsahovaly více rozšiřujících vlastností, včetně některých převzatých z UNIXu. MS-DOS od Microsoftu následovaly WINDOWS, původně běžící nad MS-DOSem, ale rokem 1995 odstartovaly samostatné WINDOWS 95, již bez podpory MS-DOSu. Další operační systém Microsoftu WINDOWS NT, kompatibilní na určité úrovni s WINDOWS 95, byl napsán kompletně celý jako nový systém ještě o něco dříve..

Další hlavní systém je UNIX, který dominuje pracovním stanicím a výkonným počítačům, jako jsou síťové servery. Je zejména populární na počítačích s výkonnými procesory RISC. Tyto počítače obvykle mají výkon minipočítače i když jsou určeny jednomu uživateli a je vcelku logické, že jsou vybaveny operačním systémem pro mikropočítače, jmenovitě UNIXem.

Zajímavý vývoj, který si začal budovat své místo v polovině 80. let, je růst sítí personálních počítačů používajících **síťový operační systém** a **distribuovaný operační systém**. V síťovém operačním systému si je uživatel vědom existence více počítačů a může se přihlásit na vzdálený počítač a kopírovat soubory z jednoho na druhý. Na každém počítači běží vlastní operační systém a má své vlastní lokální uživatele.

Síťové operační systémy se nijak zásadně neliší od jednoprocesorových operačních systémů. Obvykle potřebují síťové rozhraní a ovladač na jeho obsluhu, stejně jako programy, kterými se lze vzdáleně přihlásit nebo vzdáleně přistupovat k souborům, ale tyto dodatky nemění základní strukturu operačního systému.

Distribuovaný operační systém je v protikladu, uživateli se jeví jako jeden jednoprocesorový systém, i když je v dané chvíli složen z více počítačů a procesorů. Uživatel se nezajímá, kde se jeho program spouští a kde jsou umístěny jeho soubory; to vše je ovládáno automaticky a efektivně operačním systémem.

Skutečný distribuovaný systém vyžaduje více než jen přidání trochy kódu do jednoprocesorového systému, protože distribuovaný a centralizovaný operační systém je zásadně odlišný. Distribuovaný systém např. dovolí aplikaci, aby běžela na více procesorech ve stejném čase a tak tedy potřebuje komplexní procesorový plánovací algoritmus v souladu s požadavky na optimalizaci paralelismu.

Komunikační zpoždění v síti navíc nutí algoritmus pracovat s nekom-

pletními starými daty a dokonce s nesprávnými informacemi. Tato situace je radikálně odlišná od jednoprocesorového systému, ve kterém operační systém má kompletní informace o stavu systému.

1.3 Koncepce operačního systému

Rozhraní mezi operačním systémem a uživatelským programem je definováno jako množina „rozšiřujících instrukcí“, které operační systém vykonává. Tyto rozšířené instrukce jsou tradičně nazývány jako **systémová volání** (system calls), ačkoliv mohou být dnes implementovány mnoha způsoby. Abychom porozuměli, co operační systém dělá, musíme prozkoumat toto rozhraní z blízka. Volání dostupná v rámci rozhraní se liší mezi jednotlivými operačními systémy (i přesto, že základní koncepce je podobná).

Nyní se musíme rozhodnou, zda použít zcela obecný popis, nebo vycházet z konkrétního systému.

Je-li na výběr, zvolme raději druhý přístup. Je sice pracnější, ale nabízí lepší pohled, co operační systém doopravdy vykonává. V následujících kapitolách se podíváme blíže na systémová volání UNIXu a MINIXu (tento překlad podrobnější pohled na MINIX minimalizuje). Pro zjednodušení se budeme odkazovat hlavně na MINIX, jakožto názorný a jednoduchý systém, určený právě pro výukové účely, ale UNIXy také mají odpovídající systémová volání podle standardu POSIX. Než se podíváme na konkrétní systémová volání, měli bychom se podívat na operační systém z ptačí perfektivy trochu obecněji. Tento přehled můžeme aplikovat prakticky na všechny UNIXy.

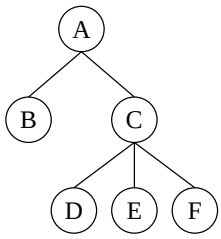
Systémová volání MINIXu se dělí zhruba na dvě vymezující kategorie: je to manipulace s procesy a se soubory. Podívejme se nyní na obě.

1.3.1 Procesy

Za klíčovou věc stavby MINIXu a všech operačních systémů je **proces**. Proces je v podstatě vykonávaný program. Ve spojitosti s každým procesem je jeho **adresový prostor** a seznam alokací paměti od minima do maxima, ve kterých proces může pracovat, tedy číst a zapisovat. Adresní prostor obsahuje proveditelný (spustitelný) program, jeho data a zásobník. Ke každému procesu také patří určitá skupina registrů, včetně čítače instrukcí (instruction pointer), zásobníku a jeho vrcholu (stack pointer) a další registry hardware potřebné pro běh programu.

Ke koncepci procesu se vrátíme v další kapitole, ale v následující chvíli je nejjednodušší pro alespoň intuitivní porozumění, zamyslet se nad systémem se sdílením času. Operační systém pravidelně rozhoduje o zastavení jednoho procesu a spuštění dalšího, např. proto, že první proces měl již více než povolené množství procesorového času v posledních vteřinách.

Pokud je proces dočasně pozastavený, musí být později znovu spuštěn



Obrázek 1: Strom procesů. Proces A má dva potomky - B a C. Proces C má tři potomky - D, E a F.

v naprosto stejném stavu, v jakém byl před zastavením. To znamená, že všechny informace o procesu musí být někde explicitně uloženy po celou dobu pozastavení. Například proces může mít otevřeno několik souborů pro čtení. S každým souborem je spojen ukazatel na aktuální pozici. Při dočasném pozastavení procesu, je nutno všechny tyto ukazatele uložit, aby při spuštění mohly funkce READ a WRITE pokračovat správně. V mnoha operačních systémech jsou všechny informace o procesu jiné, než je obsah vlastního adresního prostoru, uložený v tabulce operačního systému - **tabulce procesů** (process table), což je pole nebo zřetězený seznam s položkou pro každý existující proces zvlášť.

Pozastavený proces se tedy skládá z vlastního adresního prostoru, obvykle nazývaný **obraz procesu** (core image) a ze záznamu v tabulce procesů, kde jsou uloženy registry, včetně dalších informací.

Klíčovou činností pro správu procesů jsou systémová volání pro vytvoření a ukončení procesu. Mějme typický příklad: proces, kterému říkáme **interpret příkazů** (command interpreter) nebo **shell** přečte příkaz z terminálu. Uživatel právě napsal příkaz na kompilaci programu. Shell nyní musí vytvořit nový proces, který spustí překladač. Až proces ukončí kompilaci, pomocí systémového volání se sám ukončí.

Když proces může vytvořit jeden nebo více procesů, říkáme jim **dětské procesy**, **potomci** (child process), a tito potomci mohou opět vytvořit své potomky a rychle dostaneme stromovou strukturu procesů, např. jako na obrázku 1. Související procesy, které mají provést určitý úkol spolu často potřebují komunikovat a synchronizovat aktivity. Tuto komunikaci nazýváme **mezi-procesní komunikace - IPC** (interprocess communication) a probereme si ji později.

Pro procesy jsou i další systémová volání na realokaci množství paměti, čekání na ukončení potomka a nahrazení jednoho programu jiným.

Někdy taky potřebujeme předat informaci bežícímu procesu, který na ni vysloveně nečeká. Například proces komunikuje s jiným procesem po síti posíláním zpráv nezabezpečeným kanálem (nejde o IPC!). Abychom měli jistotu, že správa nebo odpověď byla doručena, odesílatel požaduje od svého

systému potvrzení, a pokud ho neobdrží v časovém limitu, zasílá zprávu znovu. Po nastavení časovače pokračuje program ve své činnosti.

Po uplynutí časového limitu posílá operační systém programu **signál**. Signál zapříčiní, že proces je dočasně pozastaven, ať dělal cokoliv, uloží registry na zásobník a spustí vykonávání speciální funkce pro obsluhu signálu, například přeposlání pravděpodobně ztracené zprávy. Po ukončení funkce se proces znovu spustí ve stavu, v jakém byl před signálem. Signál je programová analogie s přerušením u hardware a může být generován z mnoha příčin, tedy i vypršením časového limitu. Mnoho problémů detekovaných v hardware, jako je vykonávání ilegální instrukce, nebo porušení ochrany paměti, je konvertováno na signály zasílané procesu, který je způsobil.

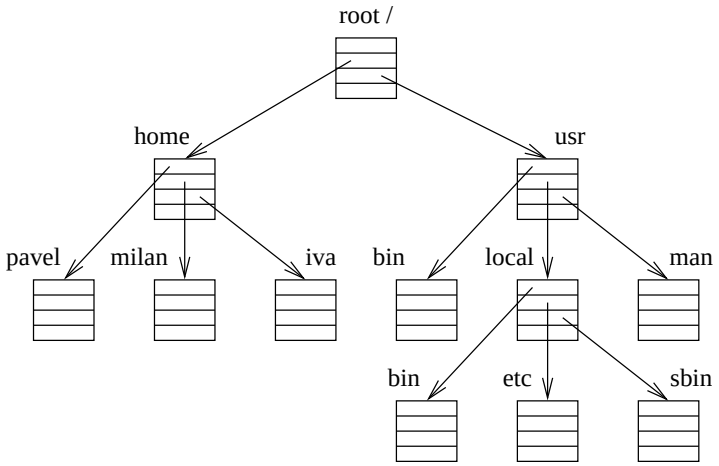
Každá osoba autorizovaná užívat MINIX má přiřazeno své **uid** (user identification) systémovým správcem. Každý proces spuštěný v MINIXu má uid osoby, která jej spustila. Potomek má stejné uid, jako jeho rodič. Jedno uid, nazývané **super-user** (root) má velký význam, může porušovat mnoho ochranných mechanismů. Na velkých systémech jen systémový správce zná heslo potřebné k tomu, být super-user, ale mnoho normálních uživatelů věnuje značné úsilí hledání chyby v systému, která by jim povolila, stát se super-user bez hesla.

1.3.2 Soubory

Druhá vymezující skupina systémových volání patří souborům. Jak bylo řečeno dříve, hlavní funkcí operačního systému je zakrýt zvláštnosti disků a jiných V/V zařízení a nabídnou programátorovi přívětivý a jasný abstraktní model souborů nezávislých na zařízení. Systémová volání jsou obvykle potřebná pro vytvoření, smazání, čtení a zápis souborů. Než bude ze souboru čteno, musí se otevřít a po ukončení čtení se musí zavřít, což je úkol systémových volání.

Pro umístění souborů používáme princip adresářů, jako metodu pro uložení souborů do skupin. Například student může mít adresář pro každý navštěvovaný předmět, další pro elektronickou poštu a domovskou WWW stránku. Potřebná systémová volání jsou pro vytvoření a smazání adresáře. Volání také zajišťují umístění souboru do adresáře a smazaní souboru z adresáře. Jednotlivé položku adresáře jsou soubory nebo adresáře. Dostáváme tak hierarchický model souborového systému jako na obrázku 2.

Hierarchie procesů i souborů je sice organizována jako strom, ale tím podobnost končí. Hierarchie procesů není příliš hluboká, obvykle 3-4 úrovně. Zatímco hierarchie souborů je obvykle 5-6 i více úrovní. Hierarchie procesů má krátkou životnost, obvykle maximálně minuty, zatím co u souborů může existovat i roky. Je taky odlišné vlastnictví i ochrana. Typicky, jen rodičovský proces smí řídit a přistupovat ke svému potomkovi, ale u souborů téměř vždy existují přístupy pro daleko větší skupinu uživatelů, než jen pro vlastníka.



Obrázek 2: Část souborového systému

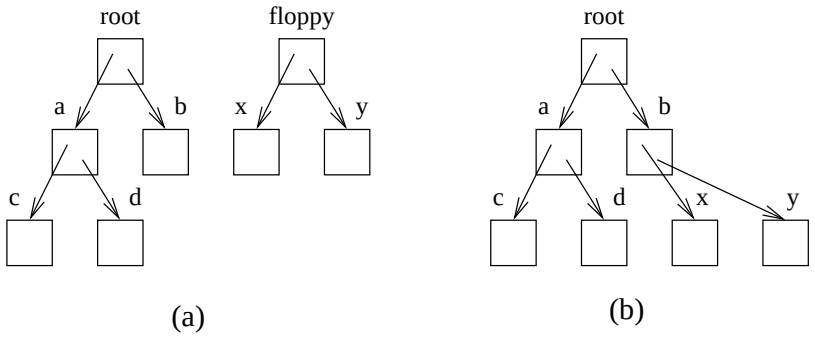
Všechny soubory v adresářové struktuře lze specifikovat uvedením **cesty** (path name) od **kořene** (root directory) adresářové hierarchie. Absolutní cesta musí obsahovat seznam adresářů, kterými musíme projít od kořene až k souboru, kde jednotlivé úrovně oddělíme lomítkem. Úvodní lomítko znamená, že cesta je absolutní, tedy od kořene.

Po celou dobu všechny procesy mají aktuální **pracovní adresář** (working directory), ve kterém se hledají soubory nezačínající lomítkem. Procesy smí změnit své pracovní adresáře systémovým voláním s určením nového pracovního adresáře.

Soubory a adresáře jsou v MINIXu chráněny přiřazením 9 bitové ochranné masky všem souborům. Ochranná maska se skládá ze tří trojic bitů, jedna pro vlastníka, druhá pro skupinu vlastníků a třetí pro všechny ostatní. Každá trojice má bit určující přístup pro čtení (**r**ead), bit pro zápis (**w**rite) a třetí pro spuštění (e**x**ecute). Tyto tři bity jsou známé jako **rwx bity**. například maska *rwxr-x–r* znamená, že vlastník může soubor číst, modifikovat i spouštět, skupina jen číst a spouštět a všichni ostatní jen spouštět. Pro adresáře *x* znamená průchod adresářem. Pomlčkou v masce říkáme, že daný přístup není povolen.

Než může být ze souboru čteno nebo zapisováno, musí se otevřít, což je okamžik, kdy se kontrolují přístupová práva. Pokud je přístup povolen, systém vrátí celé číslo, kterému říkáme **popisovač souboru** (file descriptor, handle) pro použití v následujících operacích. Pokud je přístup odmítnut, je návratovou hodnotou chybový kód.

Další důležitou vlastností MINIXu je montování souborového systému. Dříve všechny personální počítače měly jednu nebo dvě disketové mechaniky, ve kterých se vyměňovaly diskety. Pro spolehlivou manipulaci s těmito výměnnými médii MINIX povoluje souborovému systému připojit disketu do



Obrázek 3:

hlavního adresářového stromu. Mějme situaci z obrázku 3(a). Před voláním MOUNT obsahuje pevný disk hlavní adresářový strom (root file system) a disketová mechanika obsahuje disketu s jiným souborovým systémem.

Nicméně, souborový systém z diskety nemůžeme použít, protože zde není možnost, jak specifikovat cestu k souborům. MINIX nedovoluje použít cestu s uvedením názvu zařízení v začátku; to je přesně to, co operační systém musí eliminovat. Na místo toho systémové volání MOUNT dovolí souborovému systému z floppy mechaniky připojení do kořenového souborového systému kamkoliv. Obrázek 3(b) ukazuje připojení souborového systému z diskety do adresáře *b* v kořenovém adresáři. Pokud tento adresář obsahoval soubory, tak ty nebudou po dobu namontování diskety přístupné.

Další důležitou vlastností MINIXu jsou **speciální soubory** (special file, devices). Speciální soubory zajišťují přístup k zařízením jako k souborům. Tak k nim můžeme přistupovat pro čtení i zápis stejnými systémovými voláními, používanými pro práci se soubory. Jsou dva druhy speciálních souborů: blokové speciální soubory a znakové speciální soubory. Blokový speciální soubor se používá pro zařízení složené z náhodně adresovatelných bloků, jako jsou disky. Otevřením blokového speciálního souboru a čtením, řekněme bloku 4, program přistoupí přímo k bloku 4 na zařízení, bez znalosti souborového systému daného zařízení. Podobně znakové speciální soubory nahrazují tiskárny, modemy a další zařízení, která přijímají nebo vysílají proud znaků.

Poslední vlastnost kterou zmíníme v tomto přehledu, náležící souborům i procesům, to jsou **roury** (pipes). Roura je typ pseudo-souboru, který může být použit ke vzájemnému spojení dvou procesů dohromady. Rouru si lze představit jako soubor se dvěma konci, kde do jednoho konce zapisujeme a ze druhého čteme pomocí systémových volání pro práci se soubory. Samotné vytvoření roury zajišťuje systémové volání.

1.3.3 Shell

Operační systém MINIX je kód, který přináší systémová volání. Editory, překladače, assemblery a konečně i interprety příkazového řádku, nejsou součástí operačního systému, ačkoliv jsou důležité a užitečné. Ač poněkud riskujeme určitý zmatek, v této kapitole se stručně podíváme na interpret příkazů nazývaný v systému MINIX jako **shell**, který není součástí operačního systému, přesto významně používá vlastnosti operačního systému a je tedy dobrým příkladem používání systémových volání. Je to primární rozhraní mezi uživatelem u terminálu a operačním systémem.

Když se uživatel přihlásí, spustí se mu shell. Ten používá terminál jako svůj standardní vstup i výstup. jeho výstup začíná vypsáním **promptu**, který uživateli říká, že shell čeká na přijetí příkazu. Pokud nyní uživatel napíše například:

`date`,

Shell vytvoří nový proces jako svého potomka a spustí program *date*. Dokud proces běží, shell čeká na jeho ukončení. Když potomek skončí, shell vypíše nový prompt a zkouší přečíst další vstupní řádek.

Uživatel může zadat, že standardní výstup bude přesměrován do souboru, např.:

`date > file`.

Podobně může být přesměrován standardní vstup:

`sort < file1 > file2`,

kdy se spustí program *sort* a data čte z *file1* a zapisuje do *file2*.

Výstup jednoho programu může být použit jako vstup do programu dalšího spojením obou pomocí roury. Takto např.:

`cat file1 file2 | sort | lpr`.

Program přečte *cat* obsah souboru *file1* a *file2*, výstup pošle programu *sort*, který data setřídí a svůj výstup předá programu *lpr*, který data pošle na tiskárnu.

Pokud uživatel použije za příkazem znak &, nebude shell čekat na jeho ukončení a prompt se objeví okamžitě po zadání příkazu. Proces běží jako dávka na pozadí a uživatel může normálně pokračovat dál ve své práci.

1.4 Systémová volání

Když už teď víme, jak MINIX manipuluje s procesy a soubory, můžeme se podívat na rozhraní mezi operačním systémem a aplikačními programy, což je množina systémových volání.

Ačkoliv se zde budeme odvolávat na POSIX (Mezinárodní standard 9946-1), a proto taky na MINIX, většina moderních operačních systémů má systémová volání, která provádějí stejné funkce, i když se v detailech

liší. Zatím co mechanismus vyvolávání systémových volání je závislý na počítači a často musí být realizován v assembleru, knihovna funkcí zajišťuje možnost volání funkcí z programů v jazyce C.

Abychom trochu objasnili mechanismus systémových volání, podívejme se na funkci READ. Ta má tři parametry. První z nich specifikuje otevřený soubor, druhý je buffer a třetí říká, kolik bytů se má číst. Volání z jazyka C může vypadat následovně:

```
count = read( handle, &buffer, length );
```

Systémové volání vrací jako návratovou hodnotu počet přečtených bytů. Tato hodnota je obvykle stejná jako *length*, ale může být i menší, např. proto, že se při čtení dojde na konec souboru.

Pokud systémové volání nemůže být provedeno, ať už z důvodu špatných parametrů, nebo diskové chyby, je návratová hodnota -1 a chybový kód je nastaven v globální proměnné *errno*. Program by měl vždy kontrolovat výsledek systémových volání a sledovat, zda nedošlo k chybě.

MINIX má 53 systémových volání, jejich seznam uvedeme níže, uspořádaných pro přehlednost do šesti skupin.

Mimochodem, je dobré poznamenat, že povaha systémových volání je otevřená mnoha interpretacím. POSIX standard udává počet systémových volání, které systém musí podporovat, ale už neříká, zda to jsou systémová volání, funkce knihovny, nebo něco jiného. V některých případech jsou POSIX funkce v MINIXu realizovány jako podprogramy v knihovně. Řada jiných funkcí se od sebe příliš neliší a je realizována jediným systémovým voláním.

Správa procesů:

- **fork** - vytvoří nový proces - potomka, identického s rodičovský procesem,
- **waitpid** - čeká na ukončení potomka,
- **wait** - starší verze waitpid,
- **execve** - nahradí obraz procesu jiným programem,
- **exit** - ukončí aktuální proces,
- **brk** - nastaví velikost datového segmentu,
- **getpid** - ID aktuálního procesu,
- **getgrp** - ID skupiny aktuálního procesu,
- **setsid** - nová session jako ID aktuálního procesu,
- **ptrace** - pro trasování.

Signály:

- **sigaction** - definuje obsluhu signálu,
- **sigreturn** - návrat ze signálu,
- **sigprocmask** - přebere nebo změní masku signálů,
- **sigpending** - vrátí množinu blokovaných signálů,
- **sigsuspend** - vymění masku signálů a pozastaví proces,
- **kill** - zašle signál procesu,
- **alarm** - nastaví hodiny alarmu,
- **pause** - pozastaví program do dalšího signálu.

Správa souborů:

- **creat** - vytvoření souboru (zastaralá funkce),
- **mknod** - vytvoření souboru, speciálního souboru nebo adresáře,
- **open** - otevření souboru,
- **close** - uzavření souboru,
- **read** - čtení dat ze souboru,
- **write** - zápis do souboru,
- **lseek** - nastavení ukazatele v souboru,
- **stat** - status souboru,
- **fstat** - status souboru,
- **dup** - nový deskriptor pro otevřený soubor,
- **pipe** - vytvoření roury,
- **ioctl** - provádí speciální akce se souborem,
- **access** - ověří přístup k souboru,
- **rename** - přejmenuje soubor,
- **fcntl** - zamykání souborů a další operace.

Adresáře a správa souborů:

- **mkdir** - vytvoření nového adresáře,
- **rmdir** - smazání prázdného adresáře,
- **link** - nový odkaz na soubor,
- **unlink** - zmazání položky v adresáři,
- **mount** - montování souborového systému,
- **umount** - odmontování souborového systému,
- **sync** - zapíše neuložené bloky vyrovnávací paměti na disk,
- **chdir** - změna pracovního adresáře,
- **chroot** - změna kořenového adresáře.

Ochrana:

- **chmod** - změna ochranné bitové masky,
- **getuid** - UID volajícího,
- **getgid** - GID volajícího,
- **setuid** - nastaví UID,
- **setgid** - nastaví GID,
- **chown** - změna vlastníka a skupiny souboru,
- **umask** - změna masky.

Řízení času:

- **time** - čas v sekundách od 1. 1. 1970,
- **stime** - nastavení času,
- **utime** - nastavení času posledního přístupu,
- **times** - uživatelský a systémový čas procesu.