Operační systémy

Zimní semestr

Doc. Ing. Miroslav Čech, CSc.

míst. 234 – Trojanova, 2.NP

tel.: 778 532 036

E-mail: miroslav.cech@fjfi.cvut.cz

http://people.fjfi.cvut.cz/cechmiro/vyuka.html

Literatura

1. O.Čada, Operační systémy, Grada 1993
2. L. Skočovský: Principy a problémy operačního systému UNIX, Science, Praha 1993
3. Zdroje na Internetu

Operační systém – co to je?

- Operační systém je programové vybavení nezbytné pro provoz počítače.
  - Asi špatná definice – pro provoz počítače potřebuje např. sekretářka slovní procesor, který určitě do OS nepatří
- **Operační systém je správce prostředků**
  - Bude přidělovat a odebírat jednotlivé prostředky (paměť, procesor, periferie, …) programům

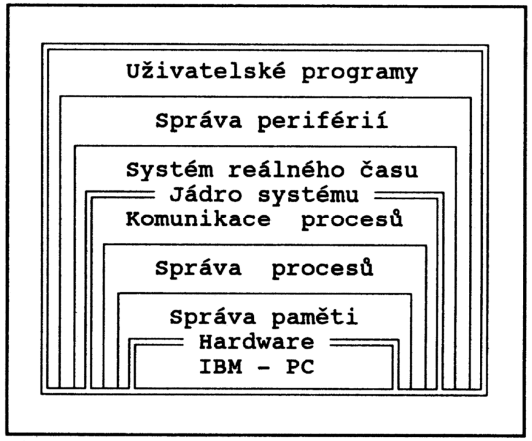
Struktura OS

- OS patří mezi nejsložitější programová díla
- Kdysi se OS psaly v Assembleru, dnes se píší ve vyšších programovacích jazycích
- Vznik jazyka C byl dán požadavkem na vývoj jazyka pro psaní operačních systémů – byl do něj převeden OS UNIX
- V současnosti se používají objektově orientované programovací jazyky (C++ apod.), pouze části, které mají být vykonávány s maximální efektivností jsou psány v Assembleru

Zásady psaní zdrojového kódu OS

- Jednoduché a krátké zdrojové texty jednotlivých funkcí – zvyšuje přehlednost programu
- Objektový přístup k psaní OS – využívání dědičnosti zvyšuje flexibilitu systémových služeb
- Vrstvená struktura – používá se pro vytváření velmi složitých programových celků – OS, programové vybavení sítí, apod. Daná vrstva využívá služeb nižší vrstvy a nabízí své služby vyšší vrstvě.

Vrstvená struktura OS



Možné uspořádání vrstev OS

Návrh OS

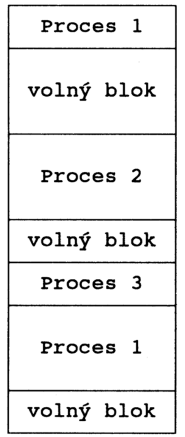
- Navrhovat OS pro běh a nezabývat se inicializací
- Inicializace probíhá pouze jednou na začátku spouštění OS, nebude vadit, když nebude příliš optimální
- Vlastní funkce důležité za běhu programu (a velmi často používané) tím pádem mohou být napsány efektivně
- Ošetření vyjímek a chybových stavů – mělo by být naprogramováno mezi prvními úkoly – výrazně to pomůže při ladění OS

Správce paměti

- Patří mezi nejdůležitější části OS
- Musí být napsán velmi pečlivě – umožní efektivní běh OS
- Hlavní úkoly správce paměti:
  - Přidělovat paměť jednotlivým procesům, když si to vyžádají
  - Udržovat informace o paměti (která část je volná, která je přidělená a komu)
  - Zařazovat paměť, kterou procesy uvolní, do volné části
  - Odebírat paměť procesům, je-li to zapotřebí

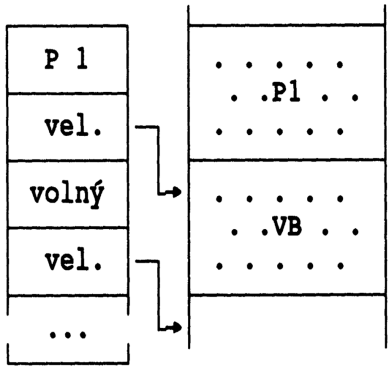
Přidělování paměti po blocích

- Každý proces musí vědět, kolik paměti bude potřebovat a musí si tuto paměť od OS vyžádat.
- OS, resp. správce paměti buď požadavek splní přidělením bloku požadované velikosti, nebo zamítne (proces musí tuto situaci sám vyřešit např. ukončením procesu nebo vyžádáním menší paměti).
- Přidělování a uvolňování paměti je dynamický proces, po určité době dojde k situaci na obrázku.



Informace o blocích paměti

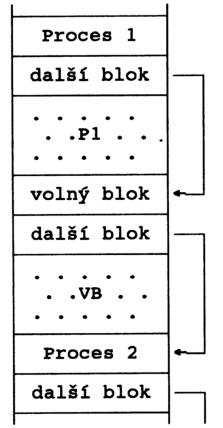
- Existuje zvláštní tabulka v OS, která udržuje informaci o obsazených blocích
- Problém je určit velikost této tabulky – pokud bude malá a bude v paměti mnoho malých bloků, zaplní se.
- Pokud bude zbytečně velká, bude mnoho položek neobsazených a bude to zabírat zbytečně paměť



1. způsob – OS udržuje zvláštní tabulku o obsazených blocích

Informace o blocích paměti

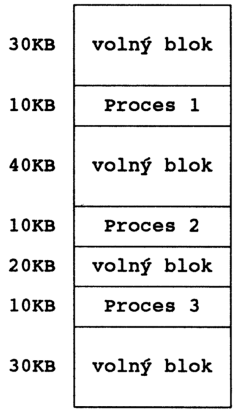
- Informace o blocích jsou „zamíchány“ v datových blocích.
- Správce paměti přidělí o trochu vetší blok, než je požadovaný.
- Zde umístí informaci ve formě spojového seznamu.
- Nebezpečí – při přepsání paměti přijde OS o informaci o jednotlivých blocích



2. Způsob – informace o blocích uvnitř datových bloků

Fragmentace paměti

- Při běhu operačního systému z důvodu požadavků na přidělení a uvolnění bloků vzniká **fragmentace** paměťového prostoru.
- Volné místo v paměti je rozděleno na množství malých bloků.
- Při požadavku na přidělení bloku paměti může dojít k situaci, kdy je dostatek volné paměti, ale není nalezen dostatečně velký blok (případ požadavku na přidělení paměťového bloku o velikosti 50KB na obrázku).

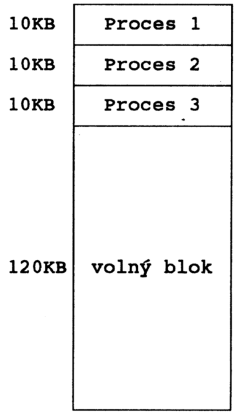


Alokační strategie

- Otázka – jak oddálit okamžik, kdy dojde k popisovanému problému?
- Možno řešit vhodnou alokační strategií přidělování bloků:
  - **First Fit** – výběr prvního dostatečně velkého bloku – postupně prochází volné bloky a z prvního dostatečně velkého požadavek vyřídí
    - Výhoda – jednoduchost a rychlost
    - Nevýhoda – nijak neomezuje fragmentaci, velké volné bloky mohou být zmenšovány požadavky na malé bloky
  - **Best Fit** – výběr bloku, jehož velikost nejlépe odpovídá požadované velikosti – projde všechny volné bloky a z těch, které jsou dostatečně velké vyhledá nejmenší, ze kterého požadavek vyřídí
    - Výhoda – zachovává velké bloky co nejdéle „nerozdrobené“
    - Nevýhoda – pomalejší, musí projít všechny volné bloky
  - **Last Fit** – varianta First fit. Požadavek se vyřídí z posledního dostatečně velkého bloku. Používá se pro zásobník (roste směrem dolů).

Defragmentace (setřesení) paměti

- V případě, že dojde k situaci, že máme dostatek volné paměti, ale nemáme dostatečně velký blok, je nutno provést **defragmentaci (setřesení)** paměti.
- Přemístíme obsazené bloky paměti tak, aby tvořily souvislou oblast (viz obrázek) a tím z volné paměti vytvoříme také jeden velký blok
- Fragmentace paměti byla odstraněna, ale jsou zde problémy:
  - Přesun dat chvíli trvá – celkem nevadí, čas může být krátký při použití DMA řadiče, také se no neprovádí často, pouze když je to nutné.
  - Procesům se změnilo umístění dat v paměti, což je velký problém



- Naprosto transparentní přístup z hlediska procesů je, když technické vybavení počítače umožňuje **překlad adres.** Pokud je ale k dispozici překlad adres, je možné implementovat tzv. **virtuální paměť**, což je nejlepší řešení (fragmentace nám vůbec nevadí) – viz později
- Pokud nemáme k dispozici překlad adres, musí procesy dodržovat určité konvence které zajistí, že přesun bloku paměti proces „nezhroutí“:
  1. Procesy musí pro přístup do paměti dodržet určité adresovací konvence, které zajistí přemístitelnost bloku (typicky se bude jednat o povinné segmentování či bázování vhodným registrem. Konkrétní řešení samozřejmé závisí na adresovacích módech použitého procesoru).
  2. Procesy musí dodržovat určité konvence na vyšší úrovni - na úrovni vlastního algoritmu. Proces může být např. povinen před přístupem do paměti zjistit dotazem u správce systému momentální adresu bloku a pak po celou dobu práce s tímto blokem paměť „zamknout“ - tj. zakázat jeho přemístění. Může přistupovat do paměťového bloku dohodnutým způsobem – např. pomocí dvojitého ukazatele (handle)
  3. Operační systém může procesu zaslat zprávu ve chvíli, kdy blok paměti přemísťuje. Proces pak na základě této zprávy přepočítá všechny své ukazatele, které do bloku míří, na správné hodnoty podle nové adresy bloku

- Metoda č. 1 je v tomto případě nejlepší, závisí však na technickém vybavení počítače (zda existují segmentové či bázové registry), může podstatným způsobem redukovat možnosti adresování v paměťových blocích (pouze adresace přes tyto registry). Musí být také přizpůsoben překladač z vyšších programovacích jazyků
- Metoda č. 2 klade nejmenší nároky na správce paměti, je však velmi náročná na dodržování z hlediska programátora aplikace – nazýváme ji kooperativní metoda (procesy resp. programátor musí s OS kooperovat).
- Metoda č. 3 může být vhodná jako doplněk k metodě č.1 pro programy, které musí pracovat z nějakých důvodů s absolutními adresami – typicky se jedná o tzv. ovladače periferních zařízení (nejsou to tedy typické aplikační programy).

Dvojitý ukazatel - handle

- Handle je vlastně dvojitý ukazatel; chceme-li tedy pracovat s blokem paměti obsahujícím např. celá čísla, deklarujeme

```
int **int_handle;
```

- Podobně můžeme samozřejmě vytvořit handle na libovolný typ, včetně složených typů. Handle můžeme i přetypovat; práce s ním je velmi podobná práci s normálním ukazatelem, až na to, že potřebujeme vždy „o hvězdičku víc“
- Jestliže máme handle deklarovaný, musíme si vyžádat od operačního systému přidělení bloku paměti. Použijeme k tomu např. systémovou funkci ‘NewHandle‘, jejímž parametrem je „velikost požadovaného bloku“. Funkce vrátí buď handle přiděleného bloku, nebo nulovou hodnotu jako informaci, že není k dispozici dostatek paměti.

- Pak můžeme s pamětí volně pracovat již popsaným způsobem. Chceme-li tedy např. vynulovat první číslo v bloku paměti, můžeme zapsat

```
**int_handle = 0;
```

- Chceme-li vynulovat celý blok paměti, můžeme napsat

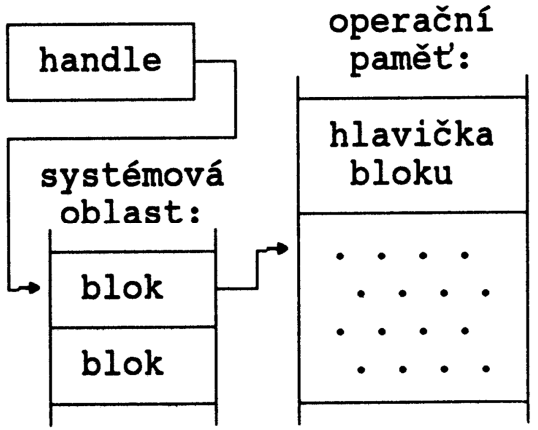
```
for (i=0; i < VELIKOST_BLOKU; i++)
       (*int_handle)[i] = 0;
```

- Zkušený programátor v jazyce C napíše jinak, podstatně efektivnějším způsobem

```
int *ptr = *int_handle;
int *end=*int_handle+VELIKOST_BLOKU;
while ( ptr < end)
       *ptr++ = 0;
```

- Nastává problém – pokud v předchozím příkladu dojde k setřesení paměti uvnitř cyklu `while`, proces zkolapsuje. Pracuje totiž s s jednoduchými ukazateli `ptr` a `end`, kde se změna umístění v paměti neprojeví.

Dvojitý ukazatel - handle



Vysvětlení funkce handle – při setřesení paměti se upraví údaje v systémové oblasti

- Ani důsledné používání dvojitého ukazatele problém neřeší. Nedá se totiž všude použít.
- Typický případ je předání údajů o paměťovém bloku do podprogramu, např. `BlockMove` pro přesun dat v paměti:

```
BlockMove(*int_handle,*int_handle+10, 10*sizeof(int));
```

- Překladač musí nejprve vyhodnotit argumenty funkce, uložit je na zásobník a pak funkci zavolat. Pokud správce paměti přesune blok mezi vyhodnocením prvního a druhého parametru, funkce zkolapsuje.
- Řešení problému - proces má k dispozici služby „HLock“ a „HUnlock“. První z těchto služeb „zamkne“ handle - to znamená, že správce paměti nesmí blok určený pomocí zamčeného handle přesunovat. Proces tedy může zamknout blok paměti a pracovat s nim zcela standardním způsobem pomocí běžných ukazatelů; po ukončení práce jej opět odemkne službou „HUnlock“

- Díky tomu je vlastně možné handle v praxi vůbec používat, musíme si však uvědomit několik velmi závažných důsledků, které více či méně degradují výkon celého systému a snižují tak výhody plynoucí z automatického přemísťováni bloků:
  - Možnost setřásat paměť jen při volání některých funkcí do jisté míry snižuje výkon celého systému. Paměť totiž pak musí být setřesena ve chvíli, kdy je to opravdu nutně zapotřebí, a všechny procesy musí čekat. Kdyby mohl operační systém setřásat paměť kdykoli, mohl by využívat volných chvil (např. když všechny procesy čekají na akci uživatele nebo na ukončení práce s diskem) k částečnému setřesení paměti; žádný čas by tak nebyl ztracen a vynucené úplné setřesení by bud nenastalo vůbec, nebo alespoň po mnohem delší době.
  - Ještě horší důsledky má možnost zamykání bloků paměti. Zamčený blok není možné setřást vůbec; pokud tedy některý proces své bloky zamyká na dlouhou dobu, narůstá fragmentace stejně jako na systému bez automatického přesunování bloků.
  - Zásadním způsobem to omezuje operační systém, který nemůže provádět potřebnou akci, když je zapotřebí

Virtuální paměť

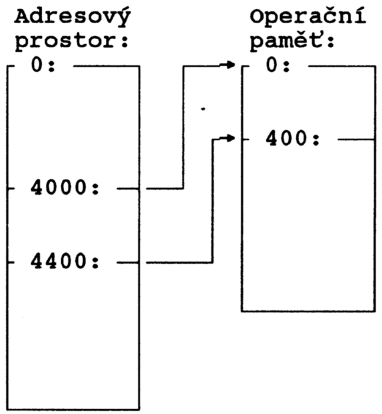
- Nejdokonalejší strategie správy paměti
- Musí být k dispozici speciální technické vybavení, tzv. **jednotka řízení paměti – Memory Management Unit MMU**
- Procesor musí být vybaven přerušovacím systémem (to dnes splňuje prakticky každý procesor) a musí být schopen zopakovat přerušený přístup do paměti (nebo i celou instrukci)
- Zopakování přístupu do paměti je základem virtualizace paměti. Pokusí-li se procesor pracovat s pamětí, která neexistuje (je virtuální), vyvolá MMU přerušení. Přerušovací podprogram neexistující paměť doplní a procesor zopakuje přístup do paměti. Protože paměť už nyní existuje, může činnost programu pokračovat nyní dále.

MMU by měla disponovat **ochranou paměti** a mechanismem **překladu adres.**

- **Ochrana paměti** – zajišťuje, aby proces nemohl přistupovat nekontrolovaně do paměti jiného procesu. MMU udržuje tabulky, ve kterých jsou informace, který úsek paměti patří kterému programu a jaká jsou přístupová práva pro procesy (jak vlastníka paměti, tak i cizí procesy). Pokud proces chce přistoupit k paměti, MMU ověří z těchto tabulek, zda má proces právo přístupu. Pokud ne, vyvolá se výjimka, v rámci které se zajistí vše potřebné (např. ukončení procesu, neboť chce použít paměť, kam nemá přístup).

- **Překlad adres** – umožňuje přiřadit libovolnému úseku v adresovém prostoru procesoru libovolné adresy ve fyzické operační paměti. Toto přiřazení se děje obyčejně pomocí tabulek, ve kterých jsou zaznamenány potřebné převody adres. Abychom odlišili obě adresy, používá se označení **logická adresa** pro adresu v adresovém prostoru procesoru a označení **fyzická adresa** pro adresu ve fyzické operační paměti. Teoreticky by překlad adres mohl fungovat pro libovolnou adresu (úsek paměti je jedna adresa), např. logická adresa 4000 by byla převedena na fyzickou adresu 0, 4001 na 1100, 4002 na 3856, atd. to by nebylo ale příliš praktické, převodní tabulky by byly příliš veliké. Zavádí se proto **stránkování**.

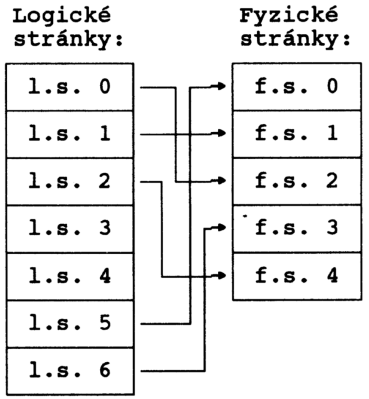
Překlad adres



Stránkování

- Požaduje, aby bloky, které se účastní překladu adres, byly složeny z tzv. stránek pevné velikosti (velmi často 4kB).
- Logický adresový prostor je tvořen stránkami (logickými), první z nich začíná na logické adrese 0, druhá na adrese <velikost_stránky>, třetí na adrese 2*<velikost_stránky>, atd.
- Stejně tak rozdělíme na jednotlivé stránky (fyzické - někdy tomu říkáme **rámce**, aby se to nepletlo) fyzickou operační paměť. První z nich začíná na fyzické adrese 0, druhá na adrese <velikost_stránky>, třetí na adrese 2*<velikost_stránky>, atd.
- Tabulky pro převod budou mít jednoduchou strukturu – pro převod jedné stránky na rámec bude v tabulce jedna položka.

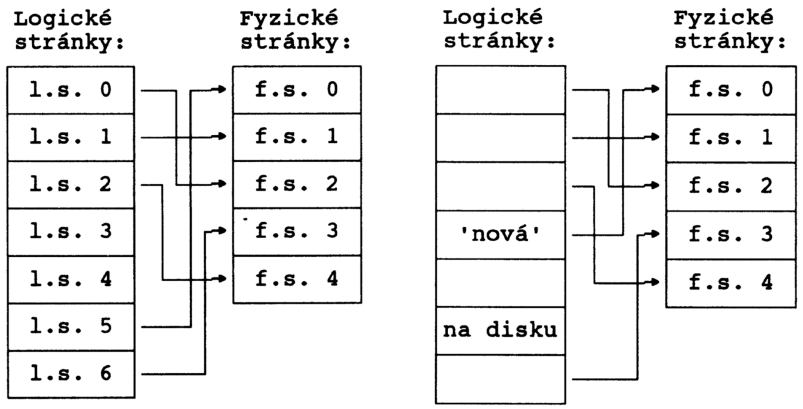
Stránkování – překlad adres



Princip virtuální paměti

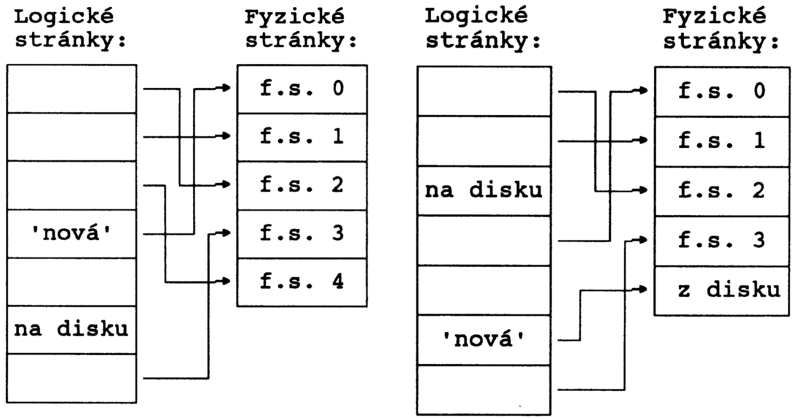
- Správce paměti má k dispozici odkládací prostor na disku.
- Pokud správce potřebuje někomu přidělit paměť a přitom není volný rámec, odebere rámec některému z procesů, uloží je do odkládacího prostoru a ve stránkových tabulkách vyznačí, že odpovídající stránka nemá přiřazen rámec a kde je odložen na disku.
- Přidělí rámec žádajícímu procesu - upraví stránkovací tabulku. Překlad adres zajistí správné adresování.
- Příklad: předpokládejme, že logická stránka číslo 3 je dosud volná a nějaký proces požaduje přidělení paměti; pro jednoduchost řekněme právě v rozsahu jedné stránky. Správce paměti přidělí logickou stránku číslo 3 a hledá odpovídající fyzickou stránku (rámec); zjistí však, že jsou všechny obsazeny. Zapíše proto obsah rámce číslo 0 do odkládacího prostoru a pozmění potřebným způsobem stránkové tabulky.

Odebrání stránky



- Pokud se původní vlastník pokusí pracovat se svou pamětí (kterou už nemá, neboť byla z fyzické paměti odložena na disk), detekuje MMU, že požadovaná adresa neexistuje. Vyvolá výjimku, tzv. **výpadek stránky**, tuto obslouží správce paměti.
- Najde odložený rámec na disku, a umístí jej někam do fyzické paměti (pokud je plná, musí uvolnit nějaký rámec – viz předcházející případ).
- Upraví stránkovací tabulky a procesor zopakuje přístup do paměti. Nyní už paměť existuje a proces může pokračovat v práci.
- Poznámka – na začátku stránka 5 měla přidělen rámec 0, nyní má přidělen rámec 4. Překlad adres ale zajistí, že vše funguje OK.

Přidělení nové stránky



Vybrání stránky pro odstranění

- Pokud potřebuje správce paměti uvolnit obsazený rámec ve fyzické paměti, nastává otázka, který má uvolnit.
- Měl by uvolnit ten rámec, který už nebude potřeba, nebo bude potřeba co nejpozději. Problém je, jak tuto informaci zjistit.
- Ukazuje se však, že je relativně rozumné vybrat rámec, který byl nejdéle nepoužitý – pokud nebyl dlouho použitý, je velká pravděpodobnost, že nebude použitý i v budoucnu – algoritmus **LRU** (**Least Recently Used**).
- Bohužel zjišťovat, který rámec byl nejdéle nepoužitý je také problém. Používá se proto jednodušší **pseudo-LRU** algoritmus, jehož výsledky se blíží výsledkům LRU.

Pseudo-LRU algoritmus

- Součástí stránkových tabulek je tzv. **Used bit** - speciální bit, který jednotka řízení paměti nastaví na jedničku při každém přístupu k odpovídající logické stránce (a tedy také k jí přiřazené stránce fyzické). To lze snadno zajistit bez jakékoli degradace výpočetního výkonu
- Operační systém pravidelně v určitém intervalu prochází stránkové tabulky a nuluje všechny „used“ bity. Nalezne-li již některý „used“bit nulový, znamená to, že tato stránka nebyla po celý interval použita (jinak by jednotka řízení paměti její „used“ bit nastavila), a je tedy dobrým kandidátem na odebrání.

Realizace virtuální paměti

- Na začátku práce má program k dispozici celý adresový prostor, ale žádnou fyzickou paměť (všechny logické stránky tedy mají nastaven přepínač, určující přiřazení fyzické stránky, na ‘neexistenci‘).
- Kdykoli se pokusí program použít nějakou adresu v logické stránce, která nemá a ani dosud neměla svůj ekvivalent v operační paměti (ze začátku tedy pokusí-li se program použít jakoukoli adresu), dojde k výpadku stránky a jednotka řízení paměti samozřejmě vyvolá přerušení. V něm se operační systém pokusí vyhledat nějakou dosud nevyužitou fyzickou stránku a přidělit ji programu. Pokud se mu to podaří, ukončí ihned přerušení a umožní tak programu pokračovat v přerušené práci - požadovaná adresa již ovšem má svůj ekvivalent v operační paměti.

- Ve chvíli, kdy operační systém nemá žádnou volnou fyzickou stránku, kterou by mohl programu přidělit, musí nějakou stránku uvolnit. K výběru fyzické stránky, která má být uvolněna, nejspíše využije strategii LRU. Původní obsah této fyzické stránky je zapotřebí zapsat do odkládací oblasti; obsah fyzické stránky se uloží na disk (tam, kde je zrovna místo), stránka se ‘staré‘ logické stránce odebere a ke ‘staré‘ logické stránce se v tabulce stránek zaznamená, kde přesně na disku je obsah stránky uložen. Pak již nic nebrání tornu, aby byla uvolněná fyzická stránka přidělena ‘nové‘ logické stránce.
- Jestliže potřebuje program použít data v logické stránce, která již dříve byla v operační paměti, ale fyzická stránka jí byla během času odebrána, proběhnou v podstatě opět předchozí dva body (nebo jen první z nich, je-li k dispozici nějaká volná fyzická stránka); jedinou změnou je to, že obsah přidělené stránky se před návratem do přerušeného programu inicializuje daty z disku, jejichž adresa byla uložena v tabulce stránek, když se fyzická stránka logické stránce odebírala podle minulého bodu.

Poznámky k virtuální paměti

- I při realizaci správce paměti pomocí virtuální paměti dochází k fragmentaci paměti. Fragmentuje se ale logický adresový prostor, který je obrovský (u současných procesorů Intel 64 TB), takže prakticky vždy budeme moci najít dostatečně velký blok pro přidělení procesu.
- Volné bloky paměti v logickém adresovém prostoru nemají svůj obraz ve fyzickém adresovém prostoru, takže nezabírají žádné místo v paměti.
- Při programování je třeba dát pozor na práci s velkými poli dat např. `double a[1000,1000]`. Pokud zpracováváme pole po řádcích a v paměti je pole uloženo po sloupcích, nevejde se do fyzické paměti z důvodu stránkování celý řádek. Při každém přístupu k prvku pole musí proběhnout výměna stránky -> velmi pomalé. Stačí zaměnit směr zpracování matice po sloupcích (v paměti budou prvky sloupce) a zpracování se výrazným způsobem zrychlí.

Ochrana paměti

- Jak už bylo řečeno, je to velmi důležitá součást operačních systémů. Vyžaduje však technické vybavení na straně procesoru.
- Moderní procesory implementují ochranu paměti pomocí speciálních tabulek (např. GDT a LDT – Global a Local Description Table u procesorů Intel), pomocí zavedení několika úrovní oprávnění – u procesorů Intel 4 úrovně a pomocí tzv. privilegovaných instrukcí.
- Úrovně oprávnění slouží k oddělení systémových a uživatelských procesů.
- Systém privilegovaných instrukcí zabraňuje vykonání některých nebezpečných instrukcí v uživatelském procesu. Pokud uživatelský proces chce vykonat privilegovanou instrukci, vyvolá se výjimka.

Procesy a multitasking

- Základní definice **multitaskingu**: multitaskový počítač (respektive operační systém, protože multitaskový operační systém lze vytvořit pro každý počítač) je takový, který umožňuje současný běh několika programů.
- Výhody multitaskingu:
  - Multitasking umožňuje kdykoli přejít k jinému programu, potřebujeme-li jej ‘na chvíli‘ použít, aniž bychom byli nuceni přerušovat rozdělanou práci.
  - Multitasking usnadňuje implementaci činností, které z principiálních důvodů musí probíhat paralelně s ostatními činnostmi počítače (dobrým příkladem je správa počítačové sítě). V multitaskovém operačním systému prostě takové činnosti zajistí samostatný proces a není pro ně zapotřebí vytvářet nový komplikovaný aparát.
  - Jednotlivé programy mohou v multitaskovém prostředí lépe kooperovat. Zatímco v jednoúlohovém prostředí si mohou programy nejvýše předávat údaje prostřednictvím souborů, mohou dva paralelně běžící programy navzájem přímo ovlivňovat svou činnost.

- Multitasking je nutnou podmínkou pro realizaci víceuživatelského systému. Má-li jediný počítač zpracovávat požadavky několika uživatelů, musí být vybaven multitaskovým operačním systémem, protože žádný z uživatelů samozřejmě nebude chtít čekat až budou ostatní hotovi.
- S multitasking umožňuje daleko lepší využití výpočetní kapacity systému. Při práci s textovým editorem počítač naprostou většinu času jen čeká až stiskneme klávesu; tuto dobu by stejně dobře mohl vyplnit prací na nějakém složitějším výpočtu (jehož výsledky budeme potřebovat až dopíšeme text). Obdobným příkladem je program, který pracuje s daty na disku např. překladač při zpracování rozsáhlého projektu. V době, kdy pracuje řadič nebo mechanika disku, se procesor může opět věnovat nějaké jiné činnosti, zatímco v klasickém jednoúlohovém systému by zahálel.

- Nevýhody multitaskingu:
  - Je-li systém špatně navržen, může současný běh většího množství programů degradovat funkci toho programu, se kterým uživatel přímo komunikuje - běh programu se neúnosně zpomalí, zvýší se doba odezvy na uživatelovy příkazy a podobně. Takovým způsobem se často chovají zvláště tzv. kooperativní systémy.
  - I multitaskový systém, který nemá nectnost popsanou v minulém bodě, má samozřejmě nějakou vlastní režii; ta se přidá k času potřebnému pro zpracování samotných programů. V krajním případě by mohlo dojít i k tzv. zahlcení systému - naprostá většina výpočetní kapacity by byla spotřebována na režii systému a programy by nebyly zpracovávány vůbec nebo téměř vůbec. I to je známka nesprávného návrhu systému; na rozdíl od minulého bodu toto nebezpečí hrozí spíše u tzv. preemptivních operačních systémů.
  - Multitaskový operační systém je samozřejmě daleko složitější a tedy i větší a dražší než systém jednoúlohový. Multitaskový operační systém často také potřebuje větší konfiguraci - rozsáhlejší operační paměť, větší kapacitu disků a výkonnější procesor.
  - Není-li multitaskový operační systém spojen s kvalitním systémem zabezpečení, zvyšuje nepříjemným způsobem riziko ztráty dat. Při zhroucení jednoúlohového systému přijdeme pouze o výsledky práce programu, který zhroucení zaviní. Jestliže však některý program ‘shodí‘ multitaskový operační systém, jsou ztraceny výsledky práce všech procesů, které v té době běžely.

Shrnutí multitaskových systémů

- Multitaskový operační systém musí být velmi pečlivě navržen, jinak je lepší zůstat u jednoúlohového systému (jehož kvalita nebo nekvalita nemá na práci počítače ani zdaleka takový vliv).
- Multitaskový operační systém se nevyplatí pro systémy, které jsou používány striktně jednoúčelově máme-li např. mikropočítač, na kterém vedeme pouze databázi zaměstnanců a nechceme po něm nic jiného (ani psaní dopisů, ani účetnictví, ani hraní počítačových her ...), byla by investice do multitaskového operačního systému zcela zbytečná.
- Multitaskový operační systém by měl být bezpečný. Jestliže tomu tak není, musíme to mít na paměti a pracovat s ním ‘opatrně‘, běží-ti nějaký důležitější program - což samozřejmě výhody systému do značné míry degraduje.
- Není pravda, že se multitaskový systém nehodí pro méně výkonné počítače, stačí, aby byl systém vhodně navržen.

Princip multitaskingu

- Pokud je k dispozici jeden procesor resp. jádra (nebo 2 až 4, což je běžné dnes), není technicky možné, aby v počítači běželo více procesů než je procesorů.
- Proto OS musí běh více procesů simulovat.
- Je třeba to provést tak, aby uživatel měl dojem, že všechny procesy běží najednou.
- OS mužeme rozdělit na následující skupiny:
  - OS s přepínáním programů (Task Switching)
  - Kooperativní multitaskové OS
  - Preemptivní multitaskové OS

Přepínání programů

- Přímý předchůdce kooperativních OS
- Je vhodný pro jednouživatelské systémy
- Uživatel může kdykoli přejít ke zpracování jiného programu, aniž by musel přerušit práci
- Možnost realizace – **vzájemné volání** – není to vlastnost OS, ale jednotlivých programů. Ty mohou umožnit spuštění jiného programu a pracovat s ním. Po ukončení programu se obnoví stav původního programu.
- Pokud ale chceme pracovat střídavě s jedním a druhým programem, tak to nejde.
- Pokud chceme pracovat s oběma programy střídavě, ale nechceme realizovat plně multitaskový OS, potřebujeme OS s tzv. **přepínáním programů**.

Existují dvě skupiny OS s přepínáním programů:

- **Omezené přepínání programů** – je možné přepínat jen mezi hlavním programem a několika speciálními programy vytvořenými výhradně pro přepínání – tzv. **pomůcky (Accessories).**
- **Neomezené přepínání programů** – umožňuje spuštění několika normálních programů a přepínání mezi nimi. Je možné přepnout do příkazového interpreteru a spustit další program. Pokud běží nějaký program, ostatní programy stojí.

Kontext

- Jak provést přerušení programu, abychom se mohli k němu po nějaké době vrátit a v běhu pokračovat?
- Musíme někam uložit kompletní stav počítače (paměť, obsah procesoru, různá zařízení, …).
- Obnovení běhu přerušeného programu by spočívalo v obnovení stavu počítače (obdoba činnosti přerušení v mikroprocesorech a běh přerušovacího podprogramu).
- Vše uložit nepřichází do úvahy – příliš mnoho dat. Stav některých zařízení ani nejde uložit (např. tiskárna) a pak se k nim vrátit

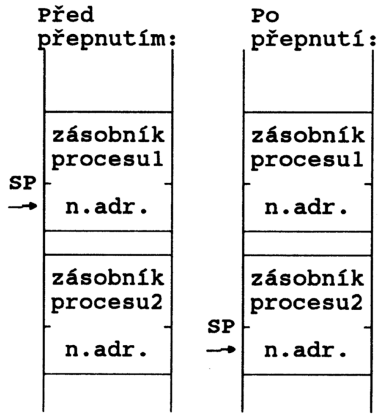
- Jestliže zajistíme, aby paměť, kterou program používá, nikdo jiný nepoužije – nemusíme ukládat.
- Před přerušením program přejde do definovaného stavu – ukončí práci se všemi periferiemi i s obrazovkou, ukončí rozpracované výpočty v pohyblivé čárce (koprocesor), nebude mít žádné údaje ve vnitřních registrech mikroprocesoru.
- Zůstane pouze nastavení zásobníku, neboli obsah registrů SS a SP u procesoru Intel 8086.
- Vytvoříme tabulku, ve které budou tato data uložena pro každý program, který je spuštěn
- Údaje v tabulce nazveme **kontext** a programu, který má v paměti kontext, budeme říkat **proces (task).**

Přepnutí kontextu

- Pokud je jeden proces aktivní a na základě nějakého signálu od uživatele je potřeba přepnou na jiný proces, OS počká, až běžící proces zavolá službu OS, kterou oznamuje že je v tzv. „přerušitelném“ stavu (každý proces je povinen to pravidelně dělat).
- Pak OS v rámci služby uloží do tabulky na místo odpovídající aktivnímu procesu adresu vrcholu zásobníku.
- Nyní zjistí z tabulky adresu vrcholu zásobníku nového procesu a zavede ji do příslušných registrů (SS a SP).
- Ukončí službu a od této chvíle běží nový proces. Došlo k **přepnutí kontextu (kontext switch).**
- Starému procesu byl **odebrán** procesor a novému procesu byl **přidělen procesor.**

- Služba systému, kterou dává proces na vědomí, že je v přerušitelném stavu, je normální podprogram se zvláštním chováním. Při přepnutí kontextu se tento podprogram vrátí na jiné místo, než byl vyvolán.
- Jelikož je přepnutí kontextu vyvoláno změnou obsahu vrcholu zásobníku, je návratová adresa, na kterou se služba vrátí, adresa v jiném procesu, který v minulosti zavolat stejnou službu a ve které mu byl odebrán procesor.
- Nyní běží tento jiný proces. Těchto přepnutí může proběhnou více.
- Po určité době se zase z nějakého procesu přepne na původní proces a přidělí se mu procesor. Teprve teď služba jakoby skončí a proces dále pokračuje.

Přepnutí kontextu



Stav zásobníku a ukazatele zásobníku

Přepínání programů

- Proces, který právě běží, musí pravidelně volat systémovou službu, kterou dává najevo, že může být přerušen (tato služba bývá kombinována s jinými službami systému. Dokud si uživatel nevyžádá přepnutí na jiný proces, nedělá tato služba nic (nebo pouze provede systémovou službu, se kterou je kombinována).
- Vyžádá-li si uživatel přepnutí procesů, zajistí zmíněná služba při nejbližším vyvolání přepnutí kontextu. Po jejím ukončení tedy již běží nový proces, a dosud aktivní proces čeká, až bude znovu aktivován.

Kooperativní multitasking

- Pokud je služba kombinována se čtením znaku s klávesnice a uživatel nechce přepnutí kontextu, bude služba čekat na stisk klávesnice (třeba velmi dlouho).
- Volný čas můžeme využít tak že přepneme na jiný proces, který něco by mohl provádět. Přitom nastavíme požadavek na aktivaci prvního procesu (který čeká na stisk klávesnice). Tomuto prvnímu procesu budeme říkat **proces na popředí (foreground process).**
- Druhý proces poběží pouze chvíli a jakmile zavolá „přerušovací službu“, dojde opět k přepnutí procesu a první proces bude čekat na příchod znaku z klávesnice. Není-li žádný k dispozici. Může se aktivovat zase jiný proces (tentýž jako předtím nebo jiný). Těmto procesům, které vypňují dobu, kdy proces na popředí na něco čeká, se říká **procesy na pozadí (backgroud process).**
- OS pracující na tomto principu se nazývá **kooperativní multitaskový OS.**

Výhody a nevýhody kooperativních OS

- Výhody:
  - Stejné jako u přepínání programů – uživatel může přepnout na jiný proces a pak se zase vrátit k původnímu procesu.
  - OS lépe využívá procesor, neboť může dobu, kdy proces čeká, vyplnit zpracování jiného procesu.
- Nevýhody – je jich hodně a závažné:
  - Zpomalení procesu na popředí – proces na popředí stojí po dobu mezi předáním řízení procesu na pozadí a zavoláním přerušovací funkce tímto procesem.
  - Nelze použít aparát na realizaci paralelních úloh (správa sítě, komunikace, …)

  - Při chybě typu nekonečná smyčka v aktivním procesu pak aktivní proces nikdy nezavolá „přerušovací službu“ a tedy nemůže dojít k přepnutí procesu – OS se zhroutí. Kooperativní OS není v principu bezpečný.
  - Při vytváření aplikací pro kooperativní multitasking musí programátoři dodržovat řadu konvencí:
    - rozdělit časově náročnou práci do kratších úseků, oddělených voláním „přerušovací služby“
    - Před voláním služby uvolnit všechny registry procesoru i koprocesoru a ukončit práci se všemi periferiemi.
    - Vytváření procesu na pozadí je velmi komplikované, často programátoři deklarují, že proces muže běžet pouze na popředí.

Preemptivní multitasking

- Základní nevýhoda kooperativního multitaskingu – nutnost, aby proces kooperoval s OS. Před přepnutím na jiný proces musí vše důležité proces uložit.
- Je třeba odstranit základní problém – nemožnost přerušit proces kdykoliv
- Jak dosáhnout toho, abychom mohli proces kdykoliv přerušit? Nemůžeme do kontextu uložit vše (stav procesoru, koprocesoru, paměti, obrazovky, vstupně-výstupních zařízení, …) byl by neúměrně veliký.

- Řešení:
  - **Zvětšíme kontext na únosnou míru – uložíme stav procesoru a koprocesoru**
  - **S ostatními prvky systému dovolíme pracovat vždy jen jedinému procesu (tzv. je vyhradíme). Pak nemusíme stav nikam ukládat. Pokud je jejich vlastník přerušen, nikdo jiný s nimi nebude pracovat a po obnovení práce je vlastník najde ve stejném (nezměněném) stavu.**
- Tzv. vyhrazení prostředků pro jediný proces má nevýhodu – může dojít k tzv. **zablokování (deadlock).** Řešení spočívá ve vyhrazení těchto prostředků speciálním systémovým procesům, tzv. **serverům.** Tyto servery ostatním procesům (**klientům**) nabízejí služby, které nepřímo umožní využití daného prostředku počítače.

- Vytvoříme-li tedy servery pro obsluhu některých prostředků, zavedeme-li povinné vyhrazení ostatních prostředků jen pro jediný proces a upravíme-li přepínání kontextu pro uložení kompletních informací o procesoru, máme vše připraveno pro implementaci preemptivního multitaskového operačního systému.
- Přepnutí kontextu za těchto podmínek totiž můžeme vyvolat kdykoli se nám zachce - v rámci kterékoli ze systémových služeb nebo třeba v rámci obslužné rutiny libovolného přerušení:
  - Je zapotřebí zajistit obsluhu sítě a odpovídající server není aktivní? Technické vybavení vyvolá přerušení a v rámci jeho obslužné rutiny operační systém přidělí síťovému serveru procesor.
  - Proces na popředí čeká na znak, takže není aktivní (běží nějaký proces na pozadí), a uživatel stiskl nějakou klávesu? Stisknutí klávesy generuje přerušení a v rámci jeho obslužné rutiny operační systém přidělí procesor procesu na popředí.
  - Pracuje již aktivní proces příliš dlouho, aniž by zavolal jakoukoli systémovou službu a umožnil tak přepnutí kontextu? Součástí technického vybavení každého počítače je časovací obvod, který generuje přerušení v pravidelných intervalech. V rámci tohoto přerušení může operační systém přidělit procesor procesu, který je právě na řadě.

Sdílení času (Time Slicing)

- Zajišťuje zdání běhu několika procesů současně.
- OS nedovolí žádnému procesu běžet déle než po určitý časový interval.
- Jestliže proces běží příliš dlouho, odebere mu OS v rámci přerušení časovače procesor a přidělí jej dalšímu procesu
- Když se vystřídají všechny procesy, dostane procesor znovu původní proces.
- Časový interval vyhrazený pro běh procesu je velmi krátký (desítky milisekund), takže rychlé střídání procesů dává iluzi současného běhu více procesů.
- Většina procesů (především interaktivních) nevyužije vymezený interval a vzdá se procesoru dříve (např. udělá potřebnou práci a bude zase čekat na nějakou periferii.

Implementace přepínání kontextu

```
typedef unsigned kontext[2];

/* ctxsw.c  */

#include  <kernel.h>
#include <proc.h>

void interrupt ctxsw (void)
{
  lastsva[0] = _SP;
  lastsva[1] = _SS;
  _SP = newsva[0];
  _SS = newsva[1];
}
```

- Implementace služby v jazyku C využívá rozšíření jazyka C. Klíčové slovo `interrupt` zajistí zápis obsahu registrů mikroprocesoru do zásobníku při vstupu do podprogramu (používá se pro psaní přerušovacích programů) a obnovení registrů při ukončení podprogramu.
- Častěji se služba `ctxsw` implementuje v assembleru z důvodu maximální efektivity. Tato systémová služba probíhá v OS velmi často.
- U procesorů Intel je možné využít systémů bran a ukládání registrů do segmentu TSS (Task Switch Segment).

Správa procesů

- Správce procesů , podobně jako správce paměti se stará o obsluhu operační paměti, se stará a sleduje procesy v systému a řídí jejich chování.
- Plánuje, který proces bude aktivován a rozhoduje, bude-li právě aktivní proces přerušen.
- Pro efektivní řešení těchto úloh si udržuje správce procesů několik **front procesů**. Každá z nich odpovídá nějaké situaci, ve které proces na něco čeká.
- Správce procesů z těchto front vybírá proces, který bude aktivován při odstranění příčiny čekání.
- Efektivní práce s frontami je základním požadavkem pro efektivitu správce procesů.

Správa front

- Fronta je objekt, na který můžeme aplikovat následující služby:
  - vytvoření fronty zajistí vznik nového objektu typu „fronta“, se kterým můžeme dále pracovat. Když jej již nepotřebujeme, můžeme jej zrušit službou zrušení fronty.
  - umístění do fronty je operace, při které se do fronty zařadí nový objekt (v našem případě proces, respektive jeho identifikační číslo, pod kterým je proces správci systému znám). Objekt nemusí být nutně zařazen na konec fronty - to závisí na konkrétním typu fronty, se kterou pracujeme.
  - odebrání z fronty je jednoduchá operace, která vrátí první objekt ve frontě a z fronty jej odstraní.

- Rozeznáváme dva typy front:
  - **Prioritní fronta** - při umístění do fronty je dalším parametrem tzv. **priorita procesu**. Správce front pak proces zařadí do fronty před všechny procesy s nižší prioritou a za všechny procesy s prioritou stejnou nebo vyšší.
  - **Fronta typu delta list** slouží k ukládání procesů, které čekají na uplynutí časového intervalu. Při umístění do fronty je dalším parametrem časový interval, po kterém má být proces opět aktivován. Správce front pak proces zařadí za všechny procesy, jejichž časový interval je kratší, a před všechny procesy s delším intervalem.

- Intervaly uložené ve frontě typu „delta list“ musí být pravidelně zkracovány (nejčastěji na základě přerušení časovače). Aby správce front nemusel frontu procházet celou, ukládá u každého procesu pouze **rozdíl jeho intervalu a intervalu předchozího**. Pak stačí, zkracuje-li pouze interval prvního procesu; po jeho vynulování může proces z fronty vyjmout a pokračovat s dalším procesem.
- Fronty mohou být implementovány jako spojové seznamy nebo tabulkou.
- Žádný proces nemůže být zároveň ve více frontách. Můžeme si vyžádat odstranění určitého procesu z fronty, aniž bychom specifikovali, ve které je.

Operace s frontami

```
/* q.h   */

isempty(q);       /* je fronta prázdná?  */
noempty(q);       /* je fronta neprázdná? */
firstkey(q);      /* nejmenší priorita ve frontě nebo čas v delta listu */
lastkey(q);       /* největší priorita ve frontě nebo čas v delta listu */
firstid(q);       /* první proces ve frontě */

enqueue(p,q);     /* zařazení procesu p na konec fronty q */
insert(p,q,key);  /*zařazení do fronty s prioritou key */
insertd(p,q,time);        /*zařazení do delta listu */
dequeue(p);       /*odstranění procesu p z fronty */
getfirst(q);      /*zjištění prvního procesu ve frontě */
getlast(q);       /*zjištění posledního procesu ve frontě */
newqueue(void);         /* vytvoření nové fronty */
```

Tabulka procesů

- V tabulce procesů jsou uloženy všechny potřebné i zdánlivě nepotřebné informace o každém procesu.
- Jde o např. stav procesu, prioritu procesu, ukládací oblast pro kontext, atd. Mohou tam být údaje pro případnou statistiku, ladící účely apod.
- Tabulka je realizována jako pole struktur, kde index do tabulky jednoznačně identifikuje proces – tzv. **PID (Process Identification)**

Tabulka procesů

```
/* process state constants */

#define   PRCURR   '\01'        /* process is currently running        */
#define   PRFREE   '\02'        /* process slot is free      */
#define   PRREADY  '\03'        /* process is on ready queue           */
#define   PRRECV   '\04'        /* process waiting for message         */
#define   PRSLEEP  '\05'        /* process is sleeping       */
#define   PRSUSP   '\06'        /* process is suspended   */
#define   PRWAIT   '\07'        /* process is on semaphore queue*/

/* process table entry */

Struct  pentry {
      unsigned char  pstate;        /* process state: PRCURR, etc.         */
      short pprio;                  /* process priority  */
      unsigned long  svarea ;       /* saved context  */
      short psem;                   /* semaphore if process waiting        */
      short pmsg;                   /* message sent to this process        */
      short phasmsg;                /* nonzero iff pmsg is valid */
      unsigned  *pbase;             /* base of run time stack  */
      unsigned  pstklen;            /* stack length              */
      unsigned  *plimit;            /* lowest extent of stack */
      char pname[PNMLEN];           /* process name   */
      short pargs;                  /* initial number of arguments         */
      void  *paddr;                 /* initial code address   */
};

extern  struct  pentry  proctab[];
```

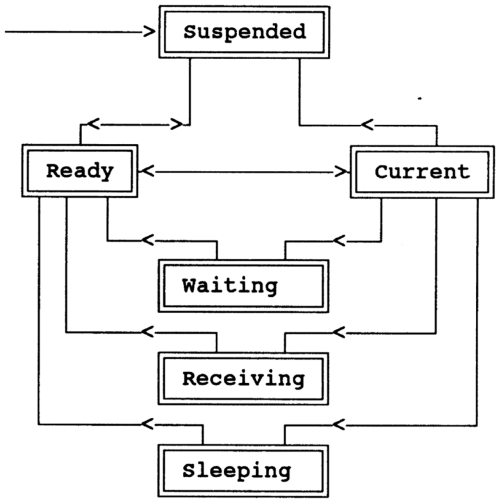
Stav procesů

- Během doby života, kdy je proces spuštěn, projde několika různými stavy – proces vznikne, pak je mu přidělen procesor a proces se střídá o procesor s dalšími procesy na základě sdílení času. Pak komunikuje s uživatelem – většinu času čeká na stisknutí klávesy. Pak si vyžádá služby serverů a čeká, než mu server pošle zprávu. Někdy tzv. „usne“ – nechá se probudit OS za nějakou dobu. Nakonec proces skončí.
- Aby správce procesů mohl efektivně procesy spravovat udržuje si u každého procesu informaci o jeho **stavu**.

- Po vytvoření se proces ocitne ve stavu **suspended.** Je to speciální stav procesu, o který nemá nikdo zájem. Není mu přidělován procesor.
- Pokud je proces připraven pro přidělení procesu, je ve stavu **ready.** Takových procesů může být více. Správce procesů udržuje tyto procesy v prioritní frontě. Pokud má být přidělen procesor, správce procesů vybere z této ready fronty první proces.
- Proces, kterému je přidělen procesor, je ve stavu **current**. Ve stavu current může být pouze jeden proces (u jednoprocesorových počítačů) nebo více (u víceprocesorových počítačů).
- Pokud je procesoru odebrán procesor, protože čeká na nějakou periferní operaci, dostane se proces do stavu **waiting**. V tomto stavu jsou procesu uloženy v tolika frontách, na kolik typů událostí se čeká.

- Čeká-li proces na přijetí zprávy od jiného procesu, dostane se do stavu **receiving**. V tomto stavu nejsou procesy umístěny v žádné frontě.
- Čeká-li proces na uplynutí určitého časového intervalu, dostane se do stavu **sleeping**. Všechny procesy v tomto stavu jsou uloženy ve společném delta listu.
- Z kteréhokoliv stavu je možné proces zrušit. Složitější systémy mají navíc stav **dead** ve kterém je proces do úplného zrušení (musí se provést relativně mnoho operací). U XINU tento stav není (XINU je příliš jednoduchý).

Stavy procesu



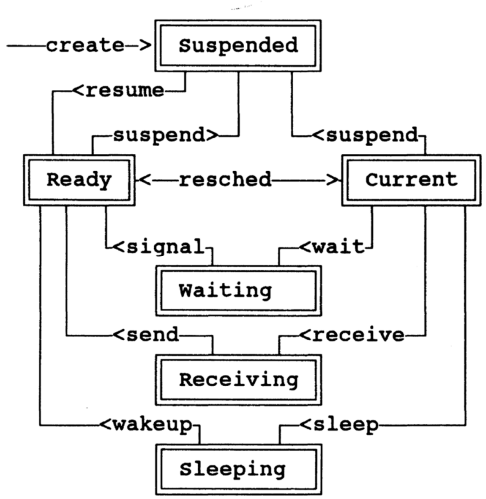
Nulový proces

- Pokud všechny procesy až na jediný čekají ve stavech suspended, waiting, sleeping nebo receiving a zbývají proces je ve stavu current (tedy ready fronta je prázdná) a tento proces ve stavu current potřebuje čekat (např. na stisknutí klávesy) nastává problém.
- Proces ze stavu current přejde do nějakého čekajícího stavu, ale není žádný proces na přidělení procesoru.
- Museli bychom přepsat funkci `ctxsw`, abychom tuto situaci vyřešili. Lepší řešení je zavést tzv. **nulový proces.**
- Operační systém při inicializaci vytvoří normální proces s velmi nízkou prioritou (nižší než je priorita všech užitečných procesů). Tento proces normálně běží, jeho nízká priorita zajistí, že je vždy poslední v ready frontě a je mu přidělen procesor jen tehdy, když ostatní procesy na něco čekají.
- Nulový proces nesmí nikdy přejít do jiného stavu než ready nebo current (nesmí zavolat nikdy systémovou službu, která by jej převedla do jiného stavu).

Plánování (Scheduling)

- Služba `ctxsw` je jádrem celého multitaskingu, ale s ní se jako uživatelé příliš nepotkáme. Tato služba je „obalená“ jinými službami, které používá jak správce procesů, tak uživatelské procesy.
- Služba `ctxsw` je obalena službou `resched` a tato je používána dalšími službami zajišťujícími multitasking.
- Přechody mezi jednotlivými stavy procesů zajišťují služby zobrazené dále.

Stavy procesu se systémovými službami



Služba `resched`

- Je základem správce procesů pro přeplánování. Vezme do úvahy stav procesů a je-li to zapotřebí, zajistí přepnutí kontextu.
- Při implementaci předpokládáme, že číslo PID aktivního procesu OS zná – je umístěno v globální proměnné `currpid`. Proto nemusí být v tabulce procesů v položce „stav“ uložen stav current, namísto tam před zavoláním služby `resched` uložíme stav, do kterého má aktivní proces po odebrání procesu přejít.

```
/* resched.c  -  resched */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*------------------------------------------------------------------------
 * resched  --  reschedule processor to highest priority ready process
 *
 * Notes:               Upon entry, currpid gives current process id.
    Proctab[currpid].pstate gives correct NEXT state for
 *                      current process if other than PRREADY.
 *------------------------------------------------------------------------
 */
int resched( void )
{
    register struct  pentry  *optr;     /* pointer to old process entry */
    register struct  pentry   *nptr;    /* pointer to new process entry */

    if ( ( (optr= &proctab[currpid])->pstate == PRCURR)  {
        if  (lastkey(rdyqueue)<optr->pprio))
            return  (OK);
       optr->pstate = PRREADY;
       insert(currpid,rdyqueue,optr->pprio);
    }
    nptr = &proctab[ (currpid = getlast(rdyqueue)) ];
    nptr->pstate = PRCURR;                      /* mark it currently running
            */
    preempt = QUANTUM;              /* reset preemption counter          */
    lastsva = (unsigned *) & (optr->svarea);
    newsva = (unsigned *) & (nptr->svarea);
    ctxsw();
    return(OK);
}
```

Služba `resched`

- Funkce `resched` nejprve ověří, je-li vůbec zapotřebí přepínat kontext je-li „budoucí“ stav aktivního procesu stále „current“ a je-li priorita aktivního procesu větší než největší priorita procesů v ready frontě, může funkce skončit.
- Není-li tomu tak, a „budoucí“ stav je stále „current“, je nutné “přeplánovat“ - funkce přeřadí dosud aktivní proces do ready fronty.
- Dále zjistí, kterému procesu má být přidělen procesor a nastaví jeho stav na “current“. Následující příkaz nastaví čítač intervalu pro sdílení času na plnou hodnotu; dosáhne-li tento čítač při dekrementaci v obslužné rutině přerušení časovače nulové hodnoty, bude opět vyvolána funkce `resched` takže procesu bude procesor odebrán (existuje-li jiný proces se stejnou nebo vyšší prioritou).
- Pak funkce vyvolá vlastní přepnutí kontextu. To znamená, že funkce `resched` byla vyvolána v rámci starého procesu, ukončena však bude v rámci procesu nového.

Služba `ready`

- Službu `resched` zabalíme ještě do další služby – `ready`.
- Parametr PID je identifikační číslo procesu, který má být přeřazen do stavu `ready`.
- Správce procesů někdy musí zařadit do stavu ready několik procesů najednou. Aby po každém zařazení nedošlo k přeplánování, můžeme použít parametr `resch`, který udává, zda se má provest přeplánování, či ještě počkat (důvod – nevíme, zda další proces nemá vyšší prioritu). Tedy převedeme procesy do stavu ready s nastavením parametru NO a teprve poslední proces převedeme do ready fronty s parametrem `resch` YES. Teprve nyní se provede přeplánování.

/* ready.c - ready */

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*------------------------------------------------------------------------
 * ready  --  make a process eligible for CPU service
 *------------------------------------------------------------------------
 */

int  ready ( int pid, int resch)
{
     register struct   pentry   *pptr;

     (pptr = &proctab[pid])->pstate = PRREADY;
     insert(pid,rdyqueue,pptr->pprio);
     if (resch)
          resched();
     return(OK);
}
```

Služby resume a suspend

- Jak již víme, je každý čerstvě vytvořený proces ve stavu ‘suspended‘. Chceme-li jej spustit (tj. převést do stavu ‘ready‘, odkud jej při nejbližším přeplánování může vybrat správce procesů k přidělení procesoru), musíme na něj použít službu ‘resume‘. Je zřejmě, že služba ‘resume‘ hude obsahovat jen o málo více než vyvolání služby ‘ready.

/* resume.c - resume */

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 * resume  --  unsuspend a process, making it ready; return the process priority
 *------------------------------------------------------------------------
 */

int  resume  ( short pid)
{
     struct   pentry  *pptr;          /* pointer to proc. tab. Entry  */
     int   prio;                      /* priority to return   */

        if ( (pptr = &proctab[pid])->pstate != PRSUSP)
             return(SYSERR);
     prio = pptr->pprio;
     ready(pid, RESCHYES);
     return(prio);
}
```

- Funkce pouze zkontroluje, nesnažíme-li se náhodou uvolnit proces, který není suspendován a zajistí vrácení priority uvolněného procesu.
- Zajímavým detailem v implementaci této služby je použití proměnné `prio`. Uvědomme si, že služba ‘ready‘ s parametrem ‘RESCHYES‘ zajistí přeplánování - dříve, než služba skončí, se tedy může stát spousta věcí; může dojít k mnohonásobnému přepnutí kontextu a ostatní procesy mohou změnit obsah systémových tabulek. Po ukončení služby ‘ready‘ se proto již nemůžeme spolehnout na hodnotu `pptr->pprio`; namísto toho je zapotřebí zapamatovat si tuto hodnotu před vyvoláním služby ‘ready‘ v lokální proměnné a pak ji jen vrátit.

/* suspend.c - suspend */

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  suspend  --  suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */

int    suspend( short  pid)
{
      struct   pentry   *pptr;     /* pointer to proc. tab. entry   */
      int  prio;                              /* priority returned   */

      if (pid==NULLPROC ||  ((pptr= &proctab[pid])->pstate!=PRCURR && pptr->pstate!=PRREADY)
            return(SYSERR);
      prio = pptr->pprio;
      if (pptr->pstate == PRREADY) {
         dequeue(pid);
         pptr->pstate = PRSUSP;
      } else {
         pptr->pstate = PRSUSP;
         resched();
      }
      return(prio);
}
```

- Služba ‘suspend‘, která převede proces do stavu ‘suspended‘, je trochu komplikovanější - musí být totiž schopna korektně zpracovat proces v kterémkoli ze stavů ‘ready‘ nebo ‘current‘.
- Příkaz ‘if‘ na začátku funkce pouze ověřuje, nejedná-li se o nulový proces a jeli skutečně suspendovaný proces ve stavu ‘ready‘ nebo ve stavu ‘current‘. Potom služba rozliší oba případy: je-li proces ve stavu ‘ready‘, stačí jej odstranit z ready fronty (a samozřejmě změnit jeho stav na ‘suspended‘).
- Jestliže však proces právě běží (to mimochodem znamená, že proces suspendoval sám sebe), musíme provést přeplánování, kterým se procesu odebere procesor. Nyní využijeme trik služby ‘resched‘ a ve stavové informaci aktivního procesu jí předáme požadovaný budoucí stav - totiž ‘suspended‘.
- Je vhodné si uvědomit, že přeplánování samozřejmě opět zapříčiní ‘velmi dlouhé trvání‘ služby ‘resched‘ a tedy i služby ‘suspend‘. Přesně řečeno, volá-li proces službu ‘suspend‘ sám na sebe, vrátí se mu z této služby řízení teprve poté, co jej někdo opět uvolní.

Služba ‘kill‘

- Služba musí vyřadit proces ze systému. Kromě změn uvnitř správy procesů tedy musí zajistit několik dalších úkolů (jako je uvolnění paměti, ve které byl zásobník procesu).
- Další komplikací pro službu ‘kii‘ je to, že ji můžeme použít na proces v libovolném stavu. Služba proto musí stav procesu detekovat a zařídit se podle něj.
- První příkaz If pouze ověří, nesnažíme-li se zrušit neexistující proces. Pak služba ‘kill‘ sníží hodnotu globální proměnné `numproc`, ve které operační systém XINU udržuje počet procesů. Dalším krokem je uvolnění paměti, ve které byl uložen zásobník procesu.
- Zbývající příkaz ‘switch‘ je v tomto případě nejpohodlnější číst od konce:
  - Po odstranění libovolného procesu (‘default:‘) musíme označit patřičnou položku v tabulce procesů jako volnou - uložíme tedy do ní hodnotu PRFREE.
  - Jestliže byl rušený proces ve stavu ‘sleeping‘ nebo ‘ready‘, znamená to, že byl umístěn v nějaké frontě. Musíme jej tedy nejprve z fronty odebrat.
  - Pro proces, který byl ve stavu ‘waiting‘, platí totéž; navíc však musíme snížit o jedničku informaci o počtu procesů, které čekají na společnou událost (více v odstavci o semaforech).
  - Zrušení aktivního procesu. V takovém případě jen označíme položku v tabulce procesů jako volnou pomocí hodnoty PRFREE a zajistíme přeplánování. Správce procesů při něm procesu odebere procesor a od té chvíle se již o tento proces nemusíme starat, protože nikdy procesor nedostane zpátky (je zruše n, nemůže se tedy dostal do ready fronty). Z hlediska tohoto rušeného procesu tedy služba ‘resched‘ nikdy neskončí (přesněji řečeno, proces přestane existovat dříve, než by služba mohla skončit).

/* kill.c - kill */

```
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *    kill  --  kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
int kill( short  pid)
      struct   pentry *pptr;      /* points to proc. table for pid*/

      if ( pptr= &proctab[pid])->pstate==PRFREE) {
            return(SYSERR);
      --numproc;
      freemem(pptr->limit, pptr->pstklen);
      switch (pptr->pstate) {
      case PRCURR:
            pptr->pstate = PRFREE;
          resched();
           /* never return  */
      case PRWAIT:
           semaph[pptr->psem].semcnt++;
      case PRSLEEP:
      case PRREADY:
            dequeue(pid);
      default:
             pptr->pstate = PRFREE;
      }
      return(OK);
}
```

Služba Create

Je třeba zajistit:

- Vyhrazení položky v tabulce procesů a vyplnění informací
- Vyhrazení paměti pro zásobník procesu.
- Na zásobníku je nutné vytvořit záznam ‘jako by‘ byl proces přerušen službou `ctxsw` bezprostředně před jeho první instrukcí, takže po prvním přidělení procesoru se proces spustí od začátku
- Paměť pro data procesu není zapotřebí vyhrazovat - proces si ji v případě potřeby vyžádá od správce paměti sám.
- Program, který bude proces zpracovávat, musí být již uložen v operační paměti. Služba `create` tedy program nezavede, ale pouze vytvoří proces na základě již zavedeného programu.

```
/* create.c - create, newpid */

#include <kernel.h>
#include <proc.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *  create  -  create a process to start running a procedure
 *------------------------------------------------------------------------
 */

int  create( void (*procaddr)(), unsigned ssize, short priority,char *name, short nargs, unsigned *args)
      {
      int    pid;                /* stores new process id */
      struct   pentry   *pptr;               /* pointer to proc. table entry */
      int  i;
      unsigned                *saddr;   /* stack address   */

      if ( priority <1 || priority >= MAXPRIO  || ((saddr=getstk(ssize)) ==  0 ) ||
               (pid=newpid()) == SYSERR) {
      if (saddr) freemem (saddr,ssize);
           return(SYSERR);
      }
      numproc++;
      (pptr = &proctab[pid] ->pstate = PRSUSP;
      for (i=0 ; i<PNMLEN && (pptr->pname[i]=name[i])!=0 ; i++) ;
      pptr->pprio = priority;
      pptr->plimit = saddr;
      pptr->pbase = saddr + ssize - sizeof(*saddr);
      pptr->pstklen = ssize;
```

```
      pptr->pargs = nargs;
      pptr->paddr = procadr;
      saddr = pptr->pbase;   /* stack grows down */
      for (i=0 ; i<nargs ; i++)  /* machine dependent; copy args  */
            *saddr--=args[i];    /* onto created process stack */
      *(unsigned long *)--sadr=(unsigned long) INITRET;  /* push on return addres (32 bit) */
      saddr--;
      *saddr--=INITFF;  /* …push on INIT flags */
      *(unsigned long *)--saddr = (unsigned long) procaddr;  /*push on process addr (32 bit) */
      saddr--;
      for (i=0; i<4;i++)
          *saddr--=0;     /* AX – DX set to zeroes */
      *saddr--=INITES;
      *saddr--=INITDS;
      for (i=0; i<3; i++)
           *saddr--=0;     /* SI,DI,DP set to zeroes  */
      pptr->svarea= (unsigned long)++saddr;

      return(pid);
}
```

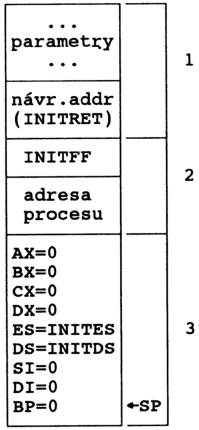
- Parametry služby create jsou: adresa programu, podle nějž bude proces pracovat, požadovaná velikost zásobníku, požadovaná priorita procesu a parametry procesu; ty služba ‘create‘ uloží na zásobník tak, že budou pro proces přístupné podle běžných konvencí jazyka C.
- Služba ‘create‘ je - podobně jako funkce ‘ctxsw‘ - do značné míry strojově závislá. Ukládání údajů na zásobník odpovídá pořadí, ve kterém registry procesoru ukládá a čte funkce typu ‘interrupt‘ použitá pro přepínání kontextu.
- Neuvádíme implementaci pomocných funkcí ‘getstk‘ a ‘newpid‘. První z nich využije služeb správce paměti pro vyhrazení paměťového bloku zadané velikosti pro zásobník a vrátí jeho adresu (nebo nulu, není-li dost volné paměti). Druhá vyhledá volné místo v tabulce procesů a vrátí odpovídající index (tedy budoucí identifikační číslo procesu), nebo nulu. je-li tabulka procesů zaplněna.
- Význam hodnoty ‘INITRET‘ - jedná se o standardní návratovou hodnotu pro případ, že proces skončí instrukcí ‘RET‘. Další hodnoty ‘INITFF‘, ‘INITES‘ a ‘INITDS‘ určují, co má být při startu procesu zavedeno do registrů procesoru FLAGS, FS a DS. Ostatní registry procesoru budou vynulovány.

- První příkaz ‘if zkontroluje, je-li možné vůbec proces vytvořit. Nejprve ověří, je-li požadovaná priorita v rozsahu přípustných priorit systému XINU; pak se pokusí alokovat blok paměti pro zásobník a nakonec zkusí vyhradit místo v tabulce procesů. Pokud se cokoli z toho nepodaří, funkce skončí a vrátí symbolickou hodnotu ‘SYSERR‘ jako informaci, že proces se nepodařilo vytvořit.
- Potom funkce ‘create‘ inkrementuje obsah globální proměnné `numproc`, ve které je uložen počet procesů v systému.
- Následujících osm řádků zdrojového textu je věnováno vyplnění položky v tabulce procesů.
- Zbytek služby ‘create‘ vytváří na zásobníku požadovaný záznam:

- Nejprve na zásobník zkopírujeme všechny parametry procesu, a pak ‘pod ně‘ uložíme adresu ‘INITRET‘. Tato část zásobníku je na obrázku označena 1. Ti, kdo znají konvence volání funkcí v jazyce C mohou potvrdit, že se jedná o standardní záznam, který na zásobník ukládá volající. Proces je tedy vlastně uměle přiveden do stejného stavu, jako kdyby byl volán z místa těsně před adresou ‘INITRET‘ se všemi parametry.
- Další úsek zásobníku, označený na obrázku 2, je standardní záznam ukládaný mikroprocesorem na zásobník při přerušení. Pro nás je důležité, že funkce typu ‘interrupt‘ v Turbo C je ukončena instrukcí pro návrat z přerušení. která tento záznam interpretuje správným způsobem (tj. uloží hodnotu ‘INITFF‘ do stavového registru a předá řízení na adresu, uloženou v záznamu spustí tedy vlastně proces.
- Zbývající úsek zásobníku, označený 3, obsahuje registry procesoru tak, jak by je uložila funkce typu ‘interrupt‘ - to samozřejmě znamená, že při ukončení funkce ‘ctxsw‘, která je typu ‘interrupt‘, budou hodnoty automaticky uloženy do správných registrů. Hodnota ukazatele zásobníku, označená na obrázku ‘SP‘, je zapsána do ukládací oblasti pro kontext v tabulce procesů; odtud ji pak funkce ‘ctxsw‘ při přepínání kontextu odebere a uloží do skutečného ukazatele zásobníku.

Stav zásobníku

1. Uloženy parametry procesu a návratová adresa (dle konvencí jazyka C)
2. Standardní záznam ukládaný mikroprocesorem na zásobník při přerušení
3. Registry mikroprocesoru, jak by je uložila funkce typu `interrupt` na zásobník



Priority a sdílení času

- XINU používá jednoduchý systém **statických priorit**. Procesu je při vytvoření přidělena priorita, která se už automaticky nemění. Může být v průběhu procesu změněna pouze na žádost uživatele (nikoli OS)
- Podobně je to i s časovým intervalem, který má proces k dispozici v rámci sdílení času. Při přeplánování je čítač času vždy nastaven na konstantní hodnotu.
- U složitějších systémů se obě hodnoty mohou měnit na základě vlastností a chování procesů, aby dosáhly co nejlepšího využití procesoru, periferních zařízení a co nejlepší průchodnosti systému. Je realizována tzv. **dynamická priorita**

Dynamická priorita

Jeden z možných (ale nejčastěji používaný) algoritmus:

- Správce procesů sleduje dobu, kterou měl proces k dispozici ve stavu current (a jak využívá interval sdílení času – tzv. **preemptivní interval**). Pokud tato doba přesáhne určitou hranici (nebo mu často musí být po vyčerpání intervalu odebrán procesor – výpočetní procesy), **sníží** se priorita tohoto procesu => nejnáročnější procesy poběží dlouho.
- Pokud proces nevyčerpává preemptivní interval (dobrovolně se vzdá procesoru, protože na něco čeká), **zvýší** mu správce procesů prioritu tím více, čím kratší dobu preemptivního intervalu vyčerpá. Takto se chovají procesy intenzivně pracující s periferiemi => reakce interaktivních procesů (jejich priorita postupně roste) je velmi rychlá.

Správa času

- Součástí správce procesů je kromě správy front i poměrně jednoduchá správa času. Jejím úkolem je umožnit procesům, aby se vzdaly možnosti přidělení procesoru po nějakou předem určenou dobu. Po uplynutí této doby musí správa času 'uspané' procesy opět aktivovat.
- Správce času převede proces, který jej požádal o 'uspání', do stavu 'sleeping', a zařadí jej do delta listu. V pravidelných intervalech (na základě přerušení časovače) pak správce sleduje delta list, není-li již načase z něj odebrat některé procesy a převést je opět do stavu 'ready'.
- Pro urychlení práce správce jsou v následujícím kódu použity proměnné `slnempty` (zda je delta list neprázdný) a `sltop` (pokud je delta list neprázdný, `sltop` ukazuje na deltu prvního procesu).
- Služba 'sleep' převede proces do stavu 'sleeping'. Patří mezi nejjednodušší služby vůbec - zařadí pouze dosud aktivní proces do delta listu ('clockqueue'), změní jeho stav na 'sleeping' a vyvolá přeplánování. Služba používá globální proměnnou `currpid`, která obsahuje číslo aktivního procesu a nastavuje globální proměnné `slnempty` a `sltop`.

/* sleep.c - sleep */

```
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * sleep  --  delay the calling process n seconds
 *------------------------------------------------------------------------
 */

void  sleep( unsigned for_time)
{
    if ( for_time != 0 ) {
            insertd(currpid,clockqueue,for_time);
            slnempty=1;
            sltop=&firstkey(clockqueue);  /*implementace ! */
            proctab[currpid].pstate=PRSLEEP;
    }
    resched();
}
```

- Čas, po který chce proces 'spát', a který předá službě 'sleep' pomocí parametru `for_time`, je určen v časových jednotkách, které závisejí na konkrétní implementaci. V nejjednodušším případě může časová jednotka odpovídat intervalu přerušení časovače; chceme-li použít delší časovou jednotku, můžeme interval prodloužit jednoduchým dělením.

/* wakeup.c - wakeup */

```
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sleep.h>

/*------------------------------------------------------------------------
 * wakeup  --  called  only when delta list is noempty !
 *------------------------------------------------------------------------
 */

void   wakeup( void )
{
    while (nonempty(clockqueue) && firstkey(clockqueue) == 0)
          ready(getfirst(clockqueue),RESCHNO);
    if ( slnempty = nonempty(clockqueue) )
        sltop = & firstkey(clockqueue);  /* ! Implementace  */
    resched();
}
```

- Služba ‘wakeup‘ není k dispozici ostatním vrstvám operačního systému, jako tomu je s ostatními službami správce procesů. Namísto toho je tato služba automaticky volána v rámci obslužné rutiny přerušení časovače.
- Služba ‘wakeup‘ zajistí převedení prvního procesu z delta listu a všech dalších procesů, které mají v delta listu svou ‘deltu‘ nulovou (to znamená, že čekaly na stejný okamžik, jako první proces) do ready fronty. Časovač samozřejmě službu ‘wakeup‘ volá až po uběhnutí potřebného času; k tomu slouží dvě globální proměnné: nenulová hodnota v proměnné `slnempty` informuje ovladač přerušení, že existuje nějaký ‘spící‘ proces, ukazatel `sltop` pak obsahuje adresu ‘delty‘ prvního procesu v delta listu - tedy zbývajícího času do chvíle ‘probuzení‘ tohoto procesu.

Synchronizace procesů

- Často různé procesy pracují nad stejnou datovou strukturou, např. fronta, sdílená paměť, apod. Pokud proces upravuje data v této datové struktuře, po určitou dobu jsou data v **nekonzistentním stavu** (některá data už byla změněna, některá ještě ne). Pokud by v této době došlo k přeplánování, nový proces, který bude chtít s datovou strukturou pracovat, by tuto datovou strukturu s největší pravděpodobností zničil (začal by používat nekonzistentní, tedy špatná data)
- Takové úseky kódu ve kterém jsou data nekonsistentní se nazývají **kritické sekce.** Je třeba zajistit, aby se dva procesy nedostaly do kritické sekce zároveň.
- Dále je třeba zajistit, aby proces počkal na pomalé periferní zařízení.
- Je třeba vytvořit aparát na tzv. **synchronizaci procesů.**
- OS používají dva mechanismy pro synchronizaci procesů – aparát **semaforů** a systém **zpráv.**

Semafory

- Základní myšlenka semaforů je poměrně jednoduchá: jestliže nesmějí dva procesy zároveň vstoupit do kritické sekce, vytvoříme programovými prostředky semafor, který postavíme na její začátek. Za normálních okolností bude na semaforu ‘zelená‘; jakmile však kolem něj projde do kritické sekce první proces, změní se barva na červenou a ostatní procesy kolem semaforu nebudou moci projít. Jestliže se o to některý proces pokusí, bude mu odebrán procesor, dokud bude na semaforu červená. Teprve ve chvíli, kdy první proces z kritické sekce odejde, změní se barva semaforu zpět na zelenou a čekající proces - je-li takový - se vrátí do stavu ‘ready‘.
- Aparát semaforů realizuje tzv. **neosobní** komunikaci (např. sdílená paměť). Jeden proces nemusí nic vědět o druhém procesu. Procesy jsou mezi sebou propojeny stejným semaforem.
- Existují dva typy semaforů:
  - **Binární semafor (Binary Semaphore)**
  - **Obecný semafor (Counting Semaphore)**

Binární semafor

- Má dva stavy (proto binární) – otevřen a uzavřen. Semafor je datová entita, např. struktura, která má dvě položky – hodnota semaforu (nabývající hodnoty 0 nebo 1 – při inicializaci nastavena na 0) a fronta semaforu, ve které jsou procesy, které čekají na semafor – jsou tedy pozastaveny (při inicializaci je fronta prázdná)
- Obyčejně bývá naprogramován pomocí služeb **Lock** a **Unlock**

Činnost služeb Lock a Unlock

- Služba ‘Lock‘ nejprve zjistí hodnotu semaforu a nastaví jej na jedničku. Tato operace musí být realizována jako nepřerušitelná, aby v době mezi testem hodnoty a nastavením na jedničku nemohl se semaforem pracovat jiný proces.
- Jestliže byl původně semafor nulový, služba ‘Lock‘ již nedělá nic dalšího a bez problémů skončí. Pokud však semafor původně obsahoval jedničku (byl uzavřen), uloží služba ‘Lock‘ PID aktivního procesu do fronty semaforu a pak jej suspenduje‘ (služba ‘Lock‘ je samozřejmě volána z aktivního procesu; ten tedy z tohoto hlediska suspenduje sám sebe).
- Služba ‘Unlock‘ nejprve ověří, je-li fronta semaforu neprázdná. Jestliže je tomu tak, vybere z ní první proces a převede jej (pomocí služby ‘resume‘) do stavu ‘ready‘. Pokud naproti tomu fronta byla prázdná, vynuluje služba ‘Unlock‘ semafor.
- Každý proces před vstupem do kritické sekce zavolá službu ‘Lock‘ a po ukončení kritické sekce službu ‘Unlock‘ (parametrem služeb je samozřejmě semafor, spojený s kritickou sekcí).

- První proces vstoupí do kritické sekce bez problémů, služba ‘Lock‘ rychle proběhne a ihned skončí. Jejím vedlejším efektem však bude nastavení semaforu na jedničku (na ‘červenou‘).
- Dokud první proces nevystoupí z kritické sekce, bude každý další proces, který zavolá službu ‘Lock‘, zařazen do fronty semaforu a suspendován. Z hlediska takového procesu vlastně služba ‘Lock‘ neskončí.
- Jakmile první proces vystoupí z kritické sekce, zavolá službu ‘Unlock‘. Ta uvolní první z procesů, čekajících ve frontě semaforu - z hlediska tohoto procesu tedy právě v tu chvíli služba ‘Lock‘ skončí a proces vstoupí do kritické sekce. Semafor zůstává ‘červený‘, takže jakýkoli další proces, který by zavolal službu ‘Lock‘, bude pozastaven a zařazen do fronty.
- Teprve ve chvíli, kdy vystoupí z kritické sekce poslední z procesů a fronta je prázdná, nastaví služba ‘Unlock‘ semafor opět na nulu.

- Binární semafory je možné použít pro ochranu kritických sekcí i v řadě dalších případů; existují však i úlohy, které jejich pomocí řešit nelze vůbec nebo jen velmi obtížně (úloha producent-konzument)
- Pokud dva procesy spolupracují způsobem, že první z nich počítá nějaké údaje a předává je druhému, který je formátuje do ‘hezké‘ podoby a prezentuje uživateli. Předpokládejme, že procesy si budou předávat data prostřednictvím sdílené paměti o velikosti 512 bytů.
- Pomocí binárních semaforů bychom mohli procesy synchronizovat tak, aby druhý proces - říkejme mu **konzument** - vždy počkal, až první proces - ten nazveme **producentem** - zapíše do sdílené paměti všech 512 bytů. Potom by opět musel čekat producent do chvíle, kdy konzument údaje z paměti odebere. Tak by se oba procesy mohly pravidelně střídat.
- Pokud bude producent generovat data v plynule, bude vše fungovat dobře. Pokud vygeneruje rychle např. 510 bytů a pak poslední 2 byty vygeneruje např. za 2 hodiny, po tuto dobu bude muset konzument čekat (i když by už mohl 510 bytů zpracovat).
- Pokud přeprogramujeme sdílenou paměť pouze na velikost 1 bytu, nastávají problémy při srovnatelné rychlosti obou procesů – bude se zvyšovat režie systému, neboť procesy budou neustále čekat jeden na druhého. Pro tyto třídy problémů se lépe hodí obecné semafory.

Obecné semafory

- Je zobecněním binárního semaforu. Hodnota obecného semaforu může nabývat nejen hodnot 0 a 1, ale libovolných celočíselných hodnot.
- Velikost hodnoty v semaforu tak může udávat, kolikrát je předplacena‘ služba ‘Lock‘, tj. kolikrát je možné ji provést, aniž bychom museli proces pozastavit. V opačném případě může velikost záporné hodnoty v semaforu určovat, kolikrát se na semafor vlastně čeká.
- Obecné semafory bývají realizovány pomocí služeb **Wait** a **Signal**. Jelikož bez problémů mohou nahradit i binární semafory, bývají v OS implementovány často jen obecné semafory.

- Služba ‘Wait‘ nejprve **sníží** hodnotu semaforu o jedničku. Je-li výsledná hodnota **menší než nula**, uloží služba ‘Waiť PID aktivního procesu do fronty semaforu, převede aktivní proces do stavu ‘waiting‘ a vyvolá přeplánování - při něm je samozřejmě procesu odebrán procesor
- Služba ‘Signal‘ nejprve **zvýší** hodnotu semaforu o jedničku. Je-li výsledná hodnota **menší nebo rovna nule**, vybere služba první proces z fronty semaforu a převede jej (pomocí služby ‘ready‘) do stavu ‘ready‘.
- Změna hodnoty semaforu a její test na velikost musí být samozřejmě provedeno jako **atomární operace** (jiný proces nesmí v tomto čase změnit nebo testovat hodnotu semaforu – typická kritická sekce !!! – u jednoprocesorových systémů zakážeme přerušení, u multiprocesorových systémů musíme řešit pomocí hardware.

Definice semaforů

```
/* sem.h */

struct   sentry  {
    signed short semcnt;    /* count for this semaphore  */
    squeue;                 /* queue            */
};
extern  struct sentry semaph[];  /* semaphores  */
```

```
/* wait.c - wait */

#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * wait  --  make current process wait on a semaphore
 *------------------------------------------------------------------------
 */

void  wait( int  sem)
{
     register struct  sentry   *sptr;
     register struct  pentry  *pptr;

     sptr= &semaph[sem];
     if (--(sptr->semcnt)<0) {
          (pptr=&proctab[currpid])->pstate = PRWAIT;
          pptr->psem=sem;     /* uložení do proctab, abychom nemuseli hledat semafor */
          enqueue(currpid,sptr->squeue);
          resched();
     }
}
```

```
/* signal.c - signal */

#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 * signal  --  signal a semaphore, releasing one waiting process
 *------------------------------------------------------------------------
 */

void  signal( int sem)
{
      register struct sentry  *sptr;

      sptr= &semaph[sem];
      if ((sptr->semcnt++) < 0)
           ready(getfirst(sptr->squeue), RESCHYES);
}
```

Řádek ve službě kill() – nyní je jasné, proč je ve službě kill() následující kód (rušíme proces, který čeká na semafor):

```
      .....
      case PRWAIT:
          semaph[pptr->psem].semcnt++;
      .....
```

Použití semaforů ve sdílené paměti

Ke sdílené paměti připojíme dva semafory; první z nich (A) inicializujeme 0 a druhý (B) inicializujeme velikostí sdílené paměti.

Kód producenta

```
for (;;) {
    char data = vytvor_data();
    wait (B);
    uloz_do_sdilene_pameti(data);
    signal (A);
}
```

Kód konzumenta:

```
for (;;) {
    char data;
    wait(A);
    data-cti_ze_sdilene_pameti();
    signal (B);
    zpracuj_data(data);
}
```

- Nyní je sdílená paměť využita nejlépe, jak to je vůbec možné. Budou-li oba procesy pracovat stejně rychle, stačí konzument odebírat data tak rychle, jak je bude producent vytvářet a k prostojům nedojde vůbec. Jestliže se například konzument trochu zdrží, umožní hodnota semaforu B producentovi generovat data bez přerušení, dokud nezaplní celou sdílenou paměť; teprve pak bude pozastaven a přinucen čekat na konzumenta. Jestliže potom poběží pomaleji naopak producent, bude konzument pozastaven teprve tehdy, až bude sdílená paměť prázdná.
- Obecné semafory mohou samozřejmě sloužit k hlídání kritických sekcí stejně snadno, jako binární semafory; (záporná) hodnota semaforu navíc udává počet procesů, které čekají na uvolnění semaforu. Proto v operačních systémech bývají obyčejně implementovány pouze obecné semafory.

Semafory v multiprocesorovém prostředí

- Připomeňme požadavek, který jsme stanovili při popisu služby ‘lock‘: mezi testem hodnoty semaforu a jeho nastavením (u obecného semaforu mezi zjištěním hodnoty a dekrementací) **nesmí v žádném případě** se semaforem pracovat nikdo jiný (jinak by mohlo dojít k současnému vstupu obou procesů do kritické sekce).
- Vytváříme-li operační systém pro počítač, osazený jediným procesorem, je to poměrně jednoduché - stačí před testem hodnoty semaforu zakázat přerušení, a po změně hodnoty semaforu jej opět povolit (Jak víme, v PC-XINU je přerušeni zakázáno dokonce po celou dobu práce kterékoli služby systému). Vážnějším problémem je implementace semaforů v operačním systému, který bude pracovat v **multiprocesorovém prostředí** (což je dnes běžné – vícejádrové procesory) - tam je nutné využít speciálních prostředků.
- Většina procesorů pro podobné účely nabízí instrukci ‘TAS‘ (‘Test And Set‘), nebo jinou instrukci s podobnou funkcí. Tato instrukce sama zajistí obě činnosti, potřebné pro realizaci službu ‘lock‘ - tj. test ‘nulovosti‘ proměnné v operační paměti a následné nastavení této proměnné na jedničku (nebo na jinou nenulovou hodnotu). Instrukce přitom pracuje s pamětí takovým způsobem, že po dobu její práce nemůže se stejnou proměnnou pracovat jiný procesor. Chceme-li tedy semafory naprogramovat ‘bezpečně‘, musíme sestoupit na úroveň assembleru a použít instrukce ‚TAS‘ (zbytek služby ‘lock‘ - tj. obsluha fronty - samozřejmě může být nadále napsán ve vyšším jazyce).

Zprávy

- Zprávy slouží pro tzv. **osobní** komunikaci. Jeden proces musí vědět o druhém.
- Jakýkoli proces může zaslat zprávu jinému procesu. Proces si může vyžádat přečtení zprávy, kterou obdržel. Jestliže dosud žádnou zprávu nedostal, OS je pozastaví do chvíle, kdy jiný proces zprávu pošle.
- Různé OS nabízejí různé typy zpráv. Systém zpráv je ideálním prostředkem pro komunikaci se servery, proto OS bývají vybaveny složitými aparáty zpráv, ve kterých se zprávy odeslané některému procesu řadí do front a obsahem zprávy může být téměř cokoli.
- XINU používá velmi jednoduchý systém zpráv. Obsahem zprávy může být jen jediné číslo a neřadí se do fronty. Pokud přijímající si zaslanou zprávu ještě nepřečetl a jiný proces mu chce poslat další zprávu, vyvolá to chybové hlášení.

/* receive.c - receive */

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return it
 *------------------------------------------------------------------------
 */

int  receive( void )
{
     struct  pentry    *pptr;

     if ((pptr = &proctab[currpid] )->phasmsg == 0) {
          pptr->pstate = PRRECV;
          resched();
     }
     pptr->phasmsg = 0;
     return(pptr->pmsg);
}
```

- Služba nejprve zjistí. má-li proces k dispozici nějakou zprávu (a tedy je položka `phasmsg` v tabulce procesů nenulová). Jestliže tomu tak není, převede proces do stavu ‘receiving‘ a zajistí přeplánování; jinak zprávu smaže a její obsah vrátí volajícímu.
- Jestliže proces neměl k dispozici žádnou zprávu, je pozastaven uvnitř funkce ‘resched‘; ta z hlediska tohoto procesu skončí teprve tehdy, až mu někdo zprávu pošle (a přeřadí jej přitom zpět do ready fronty). Je tedy zcela korektní to, že po návratu ze služby ‘resched‘ funkce čte zprávu a vrací ji volajícímu.

/* send.c - send */

```
#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  send  --  send a message to another process
 *------------------------------------------------------------------------
 */

int send( short pid, short msg)
{
     struct  pentry   *pptr;    /* receiver's proc. table addr.   */

     if ( (pptr= &proctab[pid])->pstate == PRFREE) || pptr->phasmsg != 0)
               return(SYSERR);
     pptr->pmsg = msg;                    /* deposit message              */
     pptr->phasmsg = 1;
     if (pptr->pstate == PRRECV)          /* if receiver waits, start it     */
           ready(pid, RESCHYES);

     return(OK);
}
```

- Služba nejprve ověří, zda přijímající proces vůbec existuje a jestli již nějakou zprávu nemá (XINU neumožňuje vytvářet fronty zpráv). Jestliže oba testy projdou, uloží služba zprávu do tabulky procesů a pokud cílový proces byl ve stavu ‘receiving‘, přeřadí jej do stavu ‘ready‘, aby si zprávu mohl ihned přečíst.

Zablokování (Deadlock)

Synchronizace procesů může způsobit zablokování. Semafory se používají pro vyhrazování zařízení (nutné pro realizaci preemptivního multitaskingu), kde je nebezpečí zablokování veliké. Ideální řešení je použití serverů.

```
Proces_1()
{
   wait(A);
   wait(B);
   jakasi_cinnost();
   signal(B);
   signal(A);
}
```

```
proces_2()
{
   wait(B);
   wait(A);
   jakasi_cinnost();
   signal(A);
   signal(B);
}
```

Ovladače periferií

- Počítač komunikuje s okolním světem prostřednictvím periferních zařízení. Těchto zařízení tedy samozřejmě musí využívat i jednotlivé procesy. V OS není z mnoha důvodů možné nechat procesy, aby si mezi periferiemi ‘dělaly, co je napadne‘; operační systém musí obsahovat poměrně komplikovanou správu zařízení.
- Hlavním úkolem operačního systému není ani tak zařízení přímo ovládat, jako zajistit jejich korektní přidělování jednotlivým procesům.
- Zařízení mohou být v principu tří typů:
  - vyhrazené zařízení
  - sdílené zařízení
  - společné zařízení

- **Vyhrazené zařízení** je takové zařízení, které nemůže sloužit více procesům najednou. Typickým příkladem je tiskárna: pokud bychom umožnili přístup k tiskárně více procesům najednou nebo třeba jen střídavě v kratších intervalech, bude výsledkem zmatený výstup, sestavený z promíchaných částí výstupů jednotlivých procesů.
- Operační systém proto musí obsahovat pro každé vyhrazené zařízení jeho správce; ten může využít jedné ze dvou možných technik:

Vyhrazování zařízení

- **Vyhrazování zařízení** je základní a nejjednodušší mechanismus. Při něm správce některému procesu zařízení přidělí (obvykle tomu, který s požadavkem přijde nejdříve), a ostatním procesům jeho používání nepovolí, dokud první proces zařízení neuvolní. Je to stejný princip jako kritické sekce. Zařízení je přiřazen semafor, první proces, který o zařízení požádá, tento semafor nastaví službou 'wait' na červenou a každý další proces, který požádá o přidělení téhož zařízení, bude v rámci služby 'wait' pozastaven. Teprve poté, kdy první proces svou práci se zařízením ukončí a uvolní jej pomocí služby 'signal', bude uvolněn první z procesů, které na zařízení čekají. Správce vyhrazených zařízení může zařízení přidělovat podle různých algoritmů tak, aby k zablokování nemohlo dojít. Nevýhodou tohoto řešení je to, že i ten nejlepší ze zmíněných algoritmů přidělování zařízení nepříjemným způsobem omezuje, čímž snižuje výkon celého systému.

Virtualizace

- Virtualizace vyhrazených zařízení je podobný postup, jako virtualizace paměti. Při virtualizaci paměti programy 'věří', že je k dispozici daleko více paměti než doopravdy - při virtualizaci zařízení vytvoříme procesům zdání, že namísto jediného vyhrazeného zařízení jich je k dispozici libovolně mnoho -> každý proces, který o takové zařízení požádá, může dostat svoje.

Servery

- Realizace je možná pomocí serveru, který jej bude ovládat a ostatním procesům nabídne služby, které jim využití zařízení zprostředkuje nepřímo. Virtualizace pak spočívá v tom, že proces může snadno nabízet své služby třeba všem ostatním procesům najednou bez jakéhokoli vyhrazování.
- Server se může snadno postarat o vhodné rozdělení výstupu jednotlivých procesů. Konkrétní technika samozřejmě záleží na konkrétním zařízení - na obrazovce je možné přidělit každému procesu okno a pro tiskárnu (a ji podobná zařízení) je zapotřebí použít techniky zvané **spooling.** Při ní se výstup jednotlivých procesů ukládá do souborů na disku, teprve ve chvíli, kdy proces 'tisk' zakončí (to se pozná tak, že proces uvolní 'tiskárnu'), je soubor uzavřen a předán jinému systémovému procesu, který tyto soubory postupně tiskne.

Sdílená a společná zařízení

- **Sdílená zařízení** jsou taková zařízení, která mohou svou kapacitu rozdělit na části a každou touto částí sloužit jinému procesu. Typickým případem je operační paměť nebo magnetický disk. Sdílené zařízení není zapotřebí vyhrazovat; musí však existovat správce zařízení, který se bude starat o jeho rozdělení na jednotlivé části a o přidělování těchto částí procesům.
- **Společná zařízení** mohou sloužit bez jakýchkoli problémů libovolnému počtu procesů najednou. Tato zařízení při vhodné technické realizaci jako jediná skutečně nepotřebují správce. Obvykle se jedná o jednoduchá vstupní zařízení; příkladem mohou být např. hodiny reálného času, které jsou součástí většiny počítačů, nebo mikrofon. Společná zařízení se buď obejdou zcela bez správce, nebo je jejich správce naprosto triviální

Ovladače zařízení

- Hlavním úkolem správce je zařízení přidělovat. Jsou ale důvody, proč v praxi nelze ovládání zařízení nechat na jednotlivých procesech a je třeba ovládání nechat na OS:
  - Často je zařízení ovládáno nepřímo prostřednictvím serveru. Pak OS samozřejmě zařízení ovládat musí.
  - V případě, že tomu tak není, je nutné nabídnout aplikačním programátorům vyšší úroveň ovládání zařízení, než jaké lze obvykle docílit přímým ovládáním (aby se programátoři nemuseli zabývat detaily zařízení). Je možné (a dokonce výhodnější) většinu služeb vyšší úrovně soustředit na tzv. sdílené knihovny; není však možné tak realizovat celý ovladač - důvodem je **zabezpečení systému**.
  - Ten, kdo přímo ovládá periferní zařízení, musí mít přístup k potenciálně nebezpečným službám operačního systému i technického vybavení (jako je např. mechanismus přerušení, přístup do paměti, atd.). Řada periferních zařízení navíc může při špatném řízení zapříčinit zhroucení operačního systému.

Princip ovladačů

- Ovladač by měl nabízet **služby dostatečně silné** na to, aby bylo možné využít plně všech možností zařízení. Budeme-li např. prostřednictvím ovladačů zařízení pracovat i se soubory, je nutné mít k dispozici nějaký způsob detekce konce souboru nebo zjištění jeho velikosti.
- Zároveň však by měly být všechny ovladače v maximální možné míře **navzájem podobné**, alespoň z hlediska rozhraní mezi ovladačem a procesem, který jej využívá. Pokud se podaří zajistit, že všechny ovladače budou nabízet stejné služby, je velmi snadné psát univerzální programy, které budou teprve při spuštění spojeny s konkrétním zařízením pro vstup a pro výstup.
- Je tedy zapotřebí vytvořit takovou skupinu služeb, která bude dobře sloužit pro přístup ke kterémukoli zařízení; mezi nimi přitom musí být alespoň jedna ‘speciální‘ služba, která zpřístupní neobvyklé a výjimečné vlastnosti konkrétního zařízení, samozřejmě za tu cenu, že programátor musí typ zařízení sám detekovat.

Služby ovladačů

- Služba **Init** zajistí inicializaci zařízení na začátku práce celého systému nebo po jeho restartování. Bývá obvykle bez parametrů a může vracet stav zařízení (tj. podařilo-li se jej inicializovat nebo ne).
- Řada zařízení požaduje navíc provést menší inicializaci - typicky se jedná o zařízení pro komunikaci - před zahájením vlastní komunikace bývá nutné navázat spojení, vytvořit komunikační kanál‘ kterým bude komunikace probíhat. K tomu všemu je určena služba **Open**. Zařízení, která tuto službu nepotřebují, ji mohou snadno implementovat jako prázdnou. Služba **Open** navíc může snadno sloužit k virtualizaci zařízení - její volání vytvoří nové virtuální zařízení, které bude samo nadále sloužit programu. Nejtypičtějším případem zde bývá ovladač disku; jeho služba **Open** obvykle vytvoří logické zařízení odpovídající souboru. Parametrem bývá jméno požadovaného virtuálního zařízení nebo kanálu; služba vrací nové číslo zařízení, které bude program nadále používat

- Po ukončení práce se zařízením připraveným pomocí příkazu Open je často zapotřebí vytvořený kanál opět uzavřít nebo zrušit virtuální zařízení. K tomu slouží služba **Close.** Jejím jediným parametrem bývá číslo zařízení.
- Pro vlastní výměnu dat mezi zařízením a programem jsou určeny služby **Read, Write, Getc a Putc**. První dvě předávají celé bloky dat, druhé dvě postupují po jediném bytu.
- Služba **Seek** je dokladem toho, že u velkého množství zařízení se můžeme ‘vrátit‘ a znovu si přečíst data, která jsme již jednou četli nebo můžeme nějakou skupinu údajů přeskočit a číst až za nimi. Služba sděluje ovladači relativní pozici dat, která chce program číst, v rámci celého datového bloku.
- Pro speciální případy, které není možné pokrýt dosud popsanými službami, je k dispozici služba **Cntl**. Její funkce není definována, a záleží na typu konkrétního zařízení.

Princip komunikace se zařízením

- Pro výstupní zařízení (vstupní zařízení je podobné) :
  - Ovladač na základě požadavku některého procesu zapíše data na vhodné místo v paměti a pak zařízení aktivuje, tj. předá mu příkaz ‘odešli data z té a té adresy‘.
  - Zařízení po nějakém čase data odešle a zařízení deaktivuje; pak dá ovladači na vědomí, že je možné připravit další data. Jestliže ovladač již odeslal vše, co bylo zapotřebí, je vše hotovo deaktivuje; jinak ovladač připraví další dávku dat a zařízení opět aktivuje.
  - Tento postup se opakuje, dokud není požadavek procesu zcela splněn (nebo dokud nedojde k nějaké chybě, která jeho splnění znemožní).
- Je zřejmé, že se jedná o komunikaci typu producent/konzument, (synchronizace procesů). Pokud má komunikace probíhat efektivně a bez časových ztrát, je zapotřebí použít semaforů -> ovladač musí být složen ze **dvou samostatných částí**, které spolu popsaným způsobem komunikují - první z nich slouží jako producent (budeme mu říkat ovladač), a druhá je konzumentem (obvykle je umístěna v **obslužném podprogramu přerušení**, kterým zařízení oznamuje odeslání dat).

- Ovladač běžného zařízení má dvě části. První z nich - tzv. **horní polovina** - je volána procesy (stává se tedy vlastně součástí volajícího procesu) a zajišťuje předávání údajů do sdílené paměti pro výstup a odebírání údajů ze sdílené paměti pro vstup. Horní polovina se zařízením přímo nekomunikuje, až na jedinou výjimku - aktivaci zařízení na samém začátku výstupu.
- Druhá část ovladače - tzv. **dolní polovina** - pak zajišťuje synchronizaci mezi zařízením a horní polovinou ovladače.
- Pokud by uživatelské programy volaly přímo funkce, které tvoří horní polovinu ovladačů jednotlivých zařízení, nebyl by takový přístup praktický - program by pracoval s jediným pevně zvoleným zařízením, a pro použití jiného zařízení by bylo zapotřebí jej znovu přeložit. Je potřeba to řešit jiným způsobem.

Logický systém ovladačů

- OS obsahují obyčejně tabulku zařízení, která mapuje jméno nebo identifikační číslo zařízení na jednotlivé obslužné rutiny. Ve skutečných OS je tato tabulka vytvářena dynamicky při startu systému, nebo může být měněna při zavádění a odstraňování jednotlivých ovladačů.
- XINU používá jednoduché pevné tabulky. Další zjednodušení je to, že programy se budou na jednotlivá zařízení odkazovat prostřednictvím jejich čísel; to znamená tabulku přímo indexovat číslem požadovaného zařízení.
- Ve skutečném operačním systému by místo tabulky byl použit spojový seznam instalovaných ovladačů a namísto indexace vyhledání zadaného jména v tomto seznamu.

Tabulka zařízení devtab

```
/* jména jednotlivých služeb - např. dvproc[Getc](i)) */

enum {Init,Open,Close,Read,Write,Seek,Getc,Putc,Cntl};
#define  NDVPROCS 9    /* pocet sluzeb */

struct devsw {
   int  (*dvproc[NDVPROCS])(struct *devsw, ...);  /* služby */
   int dvminor;   /* číslo konkrétního zařizení */
   int dvivec,dvovec;    /* vektory přerušení */
   void interrupt (*dviint)() ,(*dvoint)();  /* obsl. rutiny */
 };
extern struct devsw devtab[];

/* end of file */
```

- Jak je vidět, je celé zařízení popsáno poměrně jednoduchou strukturou. Položky `dvivec`, `dvovec`, `dviint` a `dvoint` určují přerušení, jejichž prostřednictvím zařízení s procesorem komunikuje, a obslužné rutiny pro tato přerušení. Operační systém tyto položky využije pouze při inicializaci, kdy zajistí. aby přerušení volalo odpovídající rutinu.
- Pole `dvproc` obsahuje adresy všech devíti služeb, o kterých jsme se zmiňovali v minulém odstavci. Pro větší přehlednost jsou přiřazena pomocí enum - u čísla jednotlivých služeb odpovídajícím identifikátorům - službu 'seek' ovladače zařízení číslo 3 tedy zavoláme příkazem

  ```
  devtab[3].dvproc[Seek](&devtab[3],pos);
  ```
- Parametry služeb jsou samozřejmě rozdílné; prvním parametrem kterékoli z nich však je ukazatel na položky tabulky 'devtab', ze které byla služba vybrána. Služby tak mají přístup ke všem polím, která mohou potřebovat.
- Poslední položkou je číslo konkrétního zařízení `dvminor`. To umožňuje řídit jediným ovladačem několik zařízení (např. osm sériových portů) - všechna taková zařízení pak budou mít obsah všech položek v tabulce zařízení identický, vyjma jediné položky právě `dvminor`. V ní bude uloženo číslo, podle kterého ovladač (tj. služby z pole 'dvproc') zjistí, se kterým zařízením má právě pracovat.

Dispečer služeb

- Originální systém XINU (stejně jako implementace ST-XINU) byl vlastně pouhou knihovnou služeb, která se spojila s přeloženými uživatelskými programy a vzniklý program se spustil jako celek. Díky tomu stačilo 'zabalit' volání služeb pro vstup a výstup do jednoduchých funkcí, které byly uživatelským programům přímo k dispozici.
- Ve skutečném operačním systému, který dokáže uživatelské programy vytvářet a zavádět za běhu, tento způsob samozřejmě nepřipadá v úvahu. Nejjednodušším řešením je využít prostředků, které nabízí procesor pro volání služeb systému (jako je např. instrukce SVC u počítačů IBM360, instrukce TRAP u mikroprocesorů Motorola nebo instrukce INT u procesorů Intel 80x86,), a připravit tzv. **dispečera služeb**. Ten přijme požadavek uživatelského programu a sám volá odpovídající službu.

```
/* h_61.c —- handler61 */
#include <conf.h>
#include <kernel.h>

extern unsigned p_seg;
void interrupt (*memo61)();
void interrupt handler61(unsigned bp,unsigned di,
                         unsigned si,unsigned ds,
                         unsigned es,unsigned dx,
                         unsigned cx,unsigned bx,
                         unsigned ax)
{
        char ah,al;
        struct devsw  *devptr;

        enable();
        ah=ax>>8;
        al=ax;
        devptr=&devtab [al];
        switch (ah) {
                case 0x00:
                    ax=devptr->dvproc[Init](devptr);
                    break;
                case 0x01:
                    ax=devptr->dvproc[Open](devptr,bx,dx);
                    break;
                case 0x02:
                    ax=devptr->dvproc[Close](devptr);
                    break;
                case 0x03:
                    ax=devptr->dvproc[Read](devptr,bx,cx);
                    break;
                case 0x04:
                    ax=devptr->dvproc[Write],bx,cx);
                    break;
                case 0x05:
                    ax=devptr->dvproc[Seek](devptr,dx);
                    break;
                case 0x06:
                    ax=devptr->dvproc[Getc](devptr);
                    break;
                case 0x07:
                    ax=devptr->dvproc[Putc],cx);
                    break;
                case 0x08:
                    ax=devptr->dvproc[Cntl](devptr,cx,bx,dx);
                    break;
                default:
                    ax=SYSERR;
        }
}
/* end of file */
```

- Implementace dispečeru je celkem samozřejmá: nejprve povolíme přerušení. které instrukce INT (v tomto případě poněkud zbytečně) zakázala. Pak rozložíme obsah registru AX do dvou čísel - vyšší, které odpovídá obsahu registru AH, je číslem požadované služby, zatímco nižší, které odpovídá obsahu registru AL, je číslem zařízení, jehož ovladač má službu provést.
- Služba se pak vyvolá uvnitř příkazu 'switch', parametry se jí předají z patřičných registrů. Návratová hodnota je uložena do registru AX a řízení se vrátí programu, který dispečer vyvolal.
- Toto řešení má obrovskou výhodu v tom, že kód služeb je součástí operačního systému a nikoli součástí programu. Navíc je možné upravovat služby (opravovat chyby, apod.) v rámci operačního systému, aniž by bylo nutno překládat uživatelské programy.
- Abychom mohli služby ovladačů zařízení pohodlně volat i z vyšších jazyků, musíme vytvořit knihovny pomocných funkcí; ty budou samozřejmě velmi jednoduché. Ukažme si příklad implementace služby 'read' na takové knihovně:

```
/* Lread.c — knihovní funkce pro volání služby 'read' */

unsigned read(int dev, void *buffer,  unsigned len)
{
    if (dev>=DEVICES)  return(BAD DEVICE);
    _AX=0x0300 I dev;
    _BX=buffer;
    _CX=len;
    geninterrupt ( 0x61);
   /* return AX */
}
/* end of file */
```

Horní polovina ovladače

- Služby horní poloviny komunikují s dolní polovinou ovladače na základě vztahu producent-konzument nad sdílenou pamětí (které v tomto kontextu obvykle říkáme **buffer**). Při výstupu dat z počítače je horní polovina ovladače v postavení producenta a dolní polovina pracuje jako konzument; při čtení dat je tomu právě naopak.
- Ovladač potřebuje pro svou činnost určitá lokální data – např. buffery, kam jsou ukládána přenášená data, ukazatele na data, apod.

Lokální data ovladače zařízení

```
#define IBUFLEN 256   /* velikost vstupniho bufferu */
#define OBUFLEN 256   /* velikost výstupního bufferu */

struct dev1 {                   /* data pro jedno zařízení */
    int ihead,itail;             /*  zacatek/konec vstupni fronty */
   char ibuf [IBUFLEN] ;        /* vstupní buffer */
   SEM isem;                    /*  vstupní semafor */
    int  ohead,otail;                    /* začátek/konec výstupní fronty */
   char obuf[OBUFLEN];          /* výstupní buffer */
   SEM osem;                    /*  výstupní semafor */
   void (*run_odev)(void);      /* aktivace výstupního zařízení */
   void (*stop_odev)(void);             /*  deaktivace výstupního zařízení */
   void (*wrt_odev) (char) ;            /*  zápis znaku na výstupní zařízení*/
  char (*rd_idev)(void);        /*  čtení znaku ze vstupního zařízení */
};
extern struct dev1 ddevs[];
```

- Oba buffery jsou realizovány jako tzv. cyklické buffery - dojdeme-li při práci s takovým bufferem na jeho konec, pokračujeme zase od jeho začátku. Pro každý buffer potom potřebujeme dva ukazatele - jeden určuje místo, kde v bufferu začínají validní data, a druhý ukazuje, kde začíná volné místo. Tyto ukazatele jsou uloženy v položkách `ihead`, `itail`, `ohead` a `otail`.
- Semafory `isem` a `osem` slouží pro synchronizaci mezí horní a dolní polovinou ovladače na základě známého mechanismu producent-konzument.
- Funkce `run_odev` povolí výstupnímu zařízení, aby přerušením hlásilo, že 'nemá co na práci'. Zařízení nemá k dispozici žádná data, a proto vygeneruje přerušení, které samozřejmě bude obslouženo dolní polovinou ovladače - ta pak data skutečně odešle.
- Funkce `stop_odev` naopak výstupnímu zařízení generování přerušení zakáže. Výstupní zařízení v takovém případě samozřejmě dokončí případný přenos dat, pokud nějaký provádí, ale jeho ukončení již nebude hlásit přerušením. Pak zůstane zařízení v klidu, dokud nebude opět zavolána služba `run_odev`.

- Funkce `wrt_odev` výstupnímu zařízení předá znak k odeslání. Zařízení znak odešle (což může trvat delší dobu), a potom generuje přerušení (má-li to povoleno předchozím voláním funkce `run_odev`).
- Funkce `rd_idev` naproti tomu přenos dat prostřednictvím vstupního zařízení nevyvolá; pouze z tohoto zařízení přečte znak, který již byl načten. Vstupní zařízení nemá nikdy zakázané přerušení; kdykoli načte nějaký znak, generuje přerušení a jeho obslužná rutina pak použije funkci `rd_idev` ke zjištění načteného znaku.
- Struktury `dev1` tvoří pole `ddevs`; v něm je každému zařízení, které má ovladač k dispozici, věnována jedna položka. Ovladač tedy prostě bude používat číslo konkrétního zařízení `dvminor` jako index do pole `ddevs`.

Služba pro čtení jednoho znaku - getc

```
/* getc.c */

#include <conf.h>
#include <driver . h>

int getc(struct devsw *dev)
{
    struct dev1 *dta=&ddevs [dev->dvminor];
    char ret,x;

    sdisable(x);
    wait(dta->isem);
    ret=dta->ibuf [dta->itail++];
    if ( dta->itail == IBUFLEN)
        dta->itail = 0;
    restore(x);
    return(ret);
}
```

- Funkce nejprve nastaví ukazatel `dta` na strukturu odpovídající požadovanému zařízení a po provedení makra `sdisable` zajistí vlastní čtení dat
- Služba 'wait', volaná na vstupní semafor, pozastaví proces na tak dlouho, dokud nebudou k dispozici nějaká data (vstupní semafor tedy musí být na počátku inicializován nulou).
- Službu 'signal' na tento semafor zavolá dolní polovina ovladače po přijetí dat. Potom bude proces znovu aktivován a data odebere ze začátku fronty ve vstupním bufferu; odebraná data uloží do proměnné `ret`. Pak je ještě zapotřebí zajistit korektní funkci cyklického bufferu případným přechodem z jeho konce na začátek.

Makra **sdisable** a **restore**

- Jsme v preemptivním multitaskovém operačním systému. Pokud by došlo k přeplánování např. mezi odebráním dat ze vstupní fronty a úpravou ukazatele (v příkazu 'if (dta- > itail ...'), mohl by nový aktivní proces zavolat sám službu 'getc' na totéž zařízení. Pak by ovšem nalezl v položce `itail` nekorektní hodnotu!
- Celý obsah služby 'getc' je tedy kritickou sekcí. Mohli bychom samozřejmě pro její ohlídání použít dalšího semaforu, ušetříme si však práci tím, že prostě zakážeme přerušení. Po dobu zákazu přerušení samozřejmě k přeplánování nemůže dojít; můžeme si to dovolit proto, že doba, po kterou bude přerušení zakázáno, bude velmi krátká - služba 'getc' se nikde a ničím nezdržuje (případné přeplánování uvnitř služby 'wait' nepřináší žádné problémy.
- Makro **sdisable** uloží do proměnné, která je jeho parametrem, informaci o tom, zda je nebo není povoleno přerušení. Pak přerušení zakáže. Makro **restore** pak ověří, je-li zapotřebí přerušení opět povolit (tedy bylo-li povoleno v okamžiku provedení makra **sdisable**), a v kladném případě tak učiní.

Makra **sdisable** a **restore**

```
#define INT_FLAG 0x200  /*poloha bitu IF v registru
                          FLAGS */

#define sdisable(x) { /* stav->x, disable */
                    x = _FLAGS;
                    disable();
                    }
#define restore(x)    /* obnov podle x  */
                  if (x & INT_FLAG)    enable();
```

Služba pro výstup znaku - putc

```
/* putc.c */

#include <conf.h>
#include <driver.h>

void putc(struct devsw *dev, char ch)
{
    struct dev1 *dta=&ddevs [dev->dvminor];
    char x;

    wait(dta->osem);
    sdisable(x);
    dta->obuf [ dta->ohead++ ] =ch;
    if ( dta->ohead == OBUFLEN)
        dta->ohead = 0;
    restore(x);
    dta->run_odev();
}
```

- Funkce **putc** nejprve nastaví ukazatel `dta` na strukturu odpovídající požadovanému zařízení a pomocí služby 'wait' a výstupního semaforu počká, dokud nebude v bufferu místo (výstupní semafor tedy musí být na počátku inicializován ne nulou nebo jedničkou, ale velikostí výstupního bufferu).
- Po návratu ze služby 'wait' v bufferu jistě místo je; proces tedy na konec fronty v bufferu uloží zadaný znak a zajistí korektní funkci cyklického bufferu případným přechodem z jeho konce na začátek.
- Nakonec funkce aktivuje zařízení službou `run_odev`; zařízení tedy při nejbližší příležitosti (pravděpodobně tedy ihned) generuje přerušení. Dolní polovina ovladače, která je obslužnou rutinou tohoto přerušení, pak data odešle.
- Uložení dat do bufferu a úprava ukazatele `ohead` je opět kritickou sekcí; tuto část kódu proto musíme ochránit před případným přeplánováním pomocí maker `sdisable` a `restore`.

Obsluha přerušení

- Pro psaní přerušovacích podprogramů využijeme možnosti překladačů pomocí klíčového slova **interrupt.** Překladač zajistí úschovu registrů na začátku podprogramu a obnovu na konci podprogramu.
- Každý procesor, aby zabránil vnořenému přerušení automaticky zakáže přerušení před vstupem do přerušovacího podprogramu -> přerušovací podprogram nesmí trvat příliš dlouho, neboť je zakázáno přerušení.
- Dalším důsledkem je to, že žádná ze služeb systému nesmí přerušení explicitně povolit. Pokud by totiž takovou službu použila obslužná rutina přerušení, mohlo by dojít ke vnořenému volání - přerušení by bylo povolené ještě před ukončením jeho obslužné rutiny. Služby systému, které musí přerušení zakazovat (protože obsahují kritické sekce) tedy musí nejprve uložit momentální stav a na konci jej obnovit tak, jak to dělají např. makra `sdisable` a `restore`.

- Ze zdrojových textů horních polovin ovladačů je jasné, že bychom potřebovali, aby dolní poloviny ovladačů volaly službu ‘signaľ. Ta však vyvolá přeplánování - je vůbec korektní volat přeplánování při zakázaném přerušení?
- První otázka může znít, nebude-li trvat přeplánování ‘moc dlouho‘. Jestliže však po přeplánování poběží dále tentýž proces, který běžel před ním, je vše v pořádku, protože bude následovat rychlé ukončení obslužné rutiny přerušení a tedy opětovné povolení přerušení.
- Stav přerušení je však součástí kontextu; dojde-li tedy v rámci přeplánování k výměně kontextu, bude stav přerušení odpovídat novému procesu. Bude-li přerušení v novém procesu povoleno, je dobře. Bude-li však přerušení v novém procesu zakázáno, znamená to, že proces se vzdal procesoru uvnitř obslužné rutiny přerušení a nyní tedy obslužnou rutinu přerušení rychle ukončí a přerušení opět povolí (je to vlastně přesně stejná situace, jako kdyby při přeplánování ke změně kontextu nedošlo).

- Poslední problém může nastat díky nulovému procesu. Nulový proces musí být vždy k dispozici v ready frontě a nesmí proto v žádném případě volat služby ‘wait‘, ‘receive‘ a podobně. Zde se však právě skrývá nebezpečí: obslužná rutina přerušení probíhá vlastně v kontextu toho procesu, který byl zrovna náhodou aktivní ve chvíli, kdy bylo přerušení vyvoláno - speciálně tedy může obslužná rutina přerušení probíhat v kontextu nulového procesu.
- Z toho vyplývá, že dolní polovina ovladače sice smí vyvolat přeplánování, musí však používat jen takové služby, které zanechají volající proces v ready frontě. To je také důvod, proč je komunikace producent-konzument v ovladači výstupního zařízení řešena poněkud netradičním způsobem. Dolní polovina ovladače je konzument znaků, a měla by proto používat služby ‘wait‘; to si však nemůžeme dovolit. Jednoduchým trikem jsme proto z dolní poloviny ovladače udělali producenta volného místa v bufferu a z horní jeho konzumenta; nyní službu ‘wait‘ používá horní polovina a vše je v naprostém pořídku.

Dolní polovina ovladače

- Dolní polovina ovladače je instalována jako obslužná rutina přerušení generovaného zařízením, které ovladač řídí. Dolní polovina ovladače vstupního zařízení je volána vždy, když vstupní zařízení přijme znak. Obslužná rutina přerušení je určena vždy pro jediné konkrétní zařízení - může tedy znát jeho číslo jako konstantu DEVXYZ.
- V rutině použije dosud nepopsanou služby ‘scount‘, jejímž parametrem je semafor. Služba vrátí hodnotu obecného semaforu. Ovladač tuto hodnotu potřebuje znát, aby zjistil, je-li ještě volné místo v bufferu.
- Ovladač přerušení výstupního zařízení vypadá velmi podobně; je jen trochu složitější, protože se musí postaral o případné ukončení přenosu.

```
/* in_iohandler.c */

#include  <conf.h>
#include  <driver.h>

#define DEV XYZ   /* číslo zařízení */

void interrupt in_ iohandler()
{
     struct dev1 *dta=&ddevs [devtab[DEV].dvminor];
     if (scount(dta->isem) == IBUFLEN)          /* buffer je plný . . . */
          rd_idev();                                    /* data přečteme a zahodíme */
     else {
          dta—>ibuf[dta->ihead++] = rd_idev( ) ;
          if (dta->ihead == IBUFLEN)
               dta->ihead = 0;
          signal (dta->isem);
     }
}
```

- Rutina nejprve nastaví ukazatel `dta` na strukturu odpovídající požadovanému zařízení. Pak zjistí, je-li v bufferu ještě místo; vzhledem k tomu, že po přijetí každého znaku zvýšila hodnotu semaforu, zatímco horní polovina ovladače při odebrání znaku hodnotu semaforu sníží, stačí nyní porovnat momentální hodnotu semaforu s velikostí bufferu. Je-li buffer plný a nikdo jej nečte, nedá se nic dělat - přijaté znaky se zahazují (trochu ‘chytřejší‘ ovladač by mohl oznámit vysílajícímu, že již nemá místo odesláním vhodného znaku pomocí ovladače výstupního zařízení; sám by naopak mohl podobný znak interpretovat a po jeho přijetí ovladač výstupního zařízení zastavit).
- Jestliže je v bufferu ještě místo, uloží na něj rutina přijatý znak a po případné úpravě ukazatele v cyklickém bufferu to oznámí horní polovině ovladače službou ‘signal’.

```
/* out_iohandler . c */

#include <conf.h>
#include <driver.h>

#define DEV XYZ     /* číslo zařízení */

void interrupt out_iohandler ( )
{
      struct dev1  *dta = &ddevs [devtab[DEV] . dvminor] ;

      if (scount(dta->osem) == OBUFLEN)       /* buffer je celý volný */
            dta->stop_odev( ) ;
       else {
            dta->wrt_odev(dta->obuf[dta->otail++] ) ;
            if (dta->otail == OBUFLEN)
                 dta->otail = 0 ;
            signal(dta->osem);
       }
}
```

- Rutina nejprve nastaví ukazatel `dta` na strukturu odpovídající požadovanému zařízení. Pak zjistí, je-li v bufferu vůbec nějaký znak; jestliže tam není, nejedná se o chybu, ale prostě o dokončení přenosu - ovladač tedy pouze deaktivuje výstupní zařízení funkcí `stop_odev`.
- Je-ti v bufferu nějaký znak, ovladač jej odešle a po případném upravení ukazatele do cyklického bufferu oznámí horní polovině získání jednoho volného místa v bufferu službou ‘signal‘.

Obrana proti zablokování (deadlock)

- Nejlepší řešení je virtualizace zařízení pomocí serveru. Může však dojít k problémům při vyčerpání systémových prostředků (např. místa na disku pro data tiskáren).
- Někdy se nevyplatí realizovat relativně složitý server. Proto existují algoritmy, jak zablokování zabránit:
  - Úplné vyhrazení prostředků
  - Hierarchické přidělování prostředků
  - Bankéřův algoritmus

Úplné vyhrazení prostředků

- Nejjednodušší strategii je nepochybně úplné vyhrazení prostředků. Správce úloh v tomto případě musí mít k dispozici informace o všech prostředcích. které bude úloha vůbec kdy potřebovat. Po dobu běhu úlohy pak tyto prostředky nepřidělí nikomu jinému (můžeme také říci, že je pro úlohu vyhradí po celou dobu jejího běhu).
- Je zřejmé, že při použití této strategie k zablokování dojít nemůže z toho prostého důvodu, že žádná úloha nikdy na přidělení nějakého zařízení nečeká. Úloha může čekat pouze na své spuštění, nejsou-li dosud k dispozici všechna zařízení, která bude požadovat; to však není nebezpečné, protože v té chvíli ještě úloha žádné zařízení vyhrazeno nemá.
- Zásadní nedostatek této metody - **obrovské snížení průchodnosti** systému. Každá úloha po celou dobu svého běhu ‘okupuje‘ všechna zařízení, která může kdy potřebovat; jestliže např. úloha deset hodin počítá a nakonec vytiskne jediné číslo jako výsledek, bude tiskárna - po celých deset hodin úlohou blokovaná
- Algoritmus je příliš defenzivní – zabraňuje nebezpečí zablokování daleko dříve, než k němu může dojít.

Hierarchické přidělování prostředků

- Nebezpečí zablokování je při křížovém vyhrazování prostředků jednotlivými procesy.
- Nebezpečí by bylo odstraněno, kdyby se v obou procesech prostředky vyhrazovaly ve stejném pořadí.
- Uspořádejme všechny prostředky do jisté hierarchie a dovolme úlohám alokovat prostředky kdykoli chtějí, ale pod jedinou podmínkou - musí prostředky požadovat ve vzrůstajícím pořadí podle dané hierarchie a uvolňovat naopak v pořadí klesajícím. V tomto případě nehrozí zablokování.

- Představme si, že v kterémkoli okamžiku pozastavíme běh systému a podíváme se, jaké prostředky mají jednotlivé úlohy vyhrazeny. Porovnáme umístění těchto prostředků v hierarchii a vybereme úlohu, která měla vyhrazen ‘nejvyšší‘ prostředek ze všech.
- Vzhledem k požadavku na postupné alokování prostředků víme, že úloha, kterou jsme si vybrali v minulém kroku, buď již nebude potřebovat žádný další prostředek, nebo bude potřebovat prostředek na vyšší hierarchické úrovni.
- Všechny prostředky na vyšší úrovni však jsou volné - připomeňme, že v prvním kroku jsme vybrali nejvyšší z přidělených prostředků. Vybraná úloha tedy nemůže být zablokována a může bez problémů pokračovat.
- V kterémkoli okamžiku tedy existuje v systému alespoň jedna úloha, která nemůže být zablokována. Po uvolnění a dalším alokování některých zařízení se samozřejmě může situace změnit; pouze však v tom smyslu, že ‘nezablokovatelná‘ bude jiná úloha.

Bankéřův algoritmus

- Nejvýhodnější by zřejmě bylo nalézt takovou strategii přidělování prostředků, která by procesy neomezovala tak dlouho, dokud by nebezpečí zablokování nehrozilo; teprve ve chvíli, kdy se objeví skutečně akutní nebezpečí zablokování by strategie prostředek nepřidělila a nechala by si jej 'v zásobě', dokud se situace nezlepší a přidělení prostředku nebude bez nebezpečí. Taková strategie skutečně existuje – tzv**. bankéřův algoritmus**
- Název strategie vznikl na základě analogie z finančního světa:

- Mějme bankéře (správce prostředků), který má k dispozici určité částky peněz v různých měnách (počty jednotlivých prostředků). Bankéř půjčuje peníze svým zákazníkům (úlohám) a ti mu je po dokončení svých záměrů opět vracejí.
- Úkolem bankéře je přitom půjčovat peníze takovým způsobem, aby nemohlo dojít k situaci, kdy by banka byla nenávratně insolventní - tj. nemohla by plnit požadavky svých zákazníků, a ti by nemohli vracet vypůjčené peníze, protože (pro nedostatek dalších peněz) by nemohli dokončit své záměry.
- Pro splnění tohoto požadavku potřebuje bankéř předem znát maximální možné požadavky každého svého zákazníka. Pak může při každé půjčce uvažovat tak, že peníze půjčí jen tehdy, když má zaručeno, že existuje alespoň jedno pořadí dalších požadavků (pořadí může bankéř ovlivnit snadno - řekne žádajícímu aby přišel později), při kterém všichni zákazníci ukončí své záměry a vrátí peníze.
- Může postupovat např. takto:

- *Musím si nejprve rozmyslet, je-li bezpečné půjčit tyto peníze. Budu tedy předpokládat, že jsem je půjčil, a zkusím, stačí-li mi zbytek ke splnění budoucích požadavků.*
- *Musím projít všechny své zákazníky a zjistit, zda je mezi nimi aspoň jeden, který mi bude moci vrátit peníze - tedy takový, jehož budoucí požadavky již nepřesáhnou množství peněz, které mi ještě zbývá. Pokud by snad žádný takový zákazník neexistoval, jistě peníze nepůjčím.*
- *Pokud někdo takový existuje, představím si, že mi již peníze vrátil. Pak obdobným způsobem zjistím, bude-li mi moci peníze vrátit některý z dalších zákazníků. Kdyby tomu tak nebylo, peníze samozřejmě také nepůjčím.*
- *Úvahu v minulém odstavci budu opakovat tak dlouho, dokud nezjistím, že peníze půjčit nemohu nebo dokud 'se nezbavím' všech zákazníků. V druhém případě je vše v pořádku a peníze mohu doopravdy půjčit.*

Detekce zablokování

- Pro přidělování virtuálních a sdílených zařízení nepoužíváme klasický 'čekací' mechanismus, ale mechanismus běžně používaný pro přidělování paměti: není-li možné požadavek uspokojit, není proces pozastaven semaforem, ale namísto toho mu operační systém vrátí informaci o tom, že se požadavek uspokojit nepodařilo (obvykle pomocí nějakého chybového kódu). Je již věcí programu samého, jak se s takovou situací vypořádá.
- Operační systém detekuje zablokování a informuje o něm operátora; ten pak může úlohy, které jej zavinily, násilně ukončit.
- Asi nejvýhodnější je kombinace obou výše uvedených technik. Pro realizaci druhé z nich je však zapotřebí, aby operační systém zablokování nějak detekoval - není dost dobře únosné, aby operátor zablokování poznal jen z toho, že se 'nějak dlouho nic neděje'.
- Princip detekce zablokování je založen na hledání uzavřené cesty v orientovaném grafu. Správce zařízení zjišťuje, jestli náhodou v pořadí 'Proces A čeká na zařízení X, které má proces B; proces B čeká na zařízení Y, které má proces C; proces C čeká ...' nemůžeme narazit znovu na proces A.

Systémový časovač

- Systémový časovač generuje přerušení v pravidelných intervalech. Operační systém tohoto přerušení obvykle využívá pro splnění dvou úkolů: sdílení času při preemptivním multitaskingu a ‘probuzení‘ procesů, které se pomocí služby ‘sleep‘ vzdaly procesoru na určitou předem danou dobu.
- Zajištění preempce je poměrně jednoduché. Obslužná rutina přerušení sleduje, kolik času ještě zbývá právě běžícími procesu; jakmile zjistí, že jeho čas vypršel, zajistí prostě přeplánování pomocí služby ‘resched‘.
- Pro probouzení ‘spících‘ procesů musí obslužná rutina přerušení zpracovávat delta list - snižuje prostě ‘deltu‘ jeho první položky. Jakmile ‘delta‘ dojde k nule, znamená to, že je zapotřebí první proces z delta listu převést do ready fronty (stejně jako všechny případné další procesy s nulovou ‘deltou‘).

- I v případě, že obslužnou rutinu přerušení časovače naprogramujeme velmi efektivně v assembleru, existuje přece jen jedna situace, kdy bude trvat nezanedbatelnou dobu dojde-li totiž v jejím rámci k přeplánování. Pro všechny běžné akce, které počítač provádí, je i tato doba dostatečně krátká, aby nenarušila žádnou funkci; je však jedna výjimka - rychlý synchronní přenos dat po blocích.
- Představme si, že máme velmi rychlou sériovou linku, po které se z hlediska programu přenášejí data ne po jednotlivých bytech. Takovým zařízením může být např. jednoduchý síťový hardware
- Pokud by uprostřed přenosu bloku po takovémto zařízení došlo k přerušení časovače, které vyvolá přeplánování, mohla bychom přijít o nějaká data; jednotlivé byty v odesílaném nebo přijímaném bloku totiž mohou jít za sebou tak rychle, že za dobu, kterou by zabrala obslužná rutina přerušení časovače a přeplánování, jich mělo přijít nebo odejít několik.

- Operační systém XINU proto obsahuje jiný mechanismus, kterému říkáme **pozastavení hodin** (anglické ‘deferred clock‘ je o něco přesnější). Jeho princip spočívá v tom, že na potřebnou dobu se nezakáže přerušení, ale zpracování jeho obslužné rutiny; operační systém si však pamatuje, jak dlouho to trvalo a po obnovení normálního stavu ihned dožene vše, co by býval měl provést, dokud byly hodiny pozastaveny (ve skutečnosti vlastně nejsou pozastaveny hodiny jako takové, ale pouze jejich obsluha).
- Služby, které zajišťují pozastavení a opětovné spuštění hodin, jsou v XINU implementovány službami **stopclk** a **strtclk**

```
void stopclk(void)
{
    defclk++;
}
```

```
void strtclk(void)
{
    char x;
    int difftime;

    sdisable(x);
    if (defclk == 0 I I --defclk > 0) {
        restore(x);
        return;
    }
    preempt -= (difftime = clkdiff);
    clkdiff = 0;
    if (slnempty) {
        int pid = firstid(clockqueue);
        int *delta;
```

```
    do {
            if (*(delta = deltaid(clockqueue,pid)) < difftime) {
                difftime -= *delta;
                *delta = 0;
            } else break;
            pid = nextpid(clockqueue,pid);
        } while (pid >= 0);

     if (pid >= 0) /* neprošli jsme celou frontu */
          *delta -= difftime;
      wakeup();
   }
   if (preempt <= 0)
         resched();
   restore(x);
}
```

- Globální proměnná `defclk` slouží podobně jako obecné semafory: ukazuje, kolikrát byla volána služba ‘stopclk‘ bez ukončení odpovídající službou ‘strtclk‘.
- Kdykoli je tedy hodnota proměnné `defclk` větší než nula, jsou hodiny pozastaveny. Ze stejného důvodu, jako je tomu u semaforů, musíme také test a nastavení proměnné `defclk` na začátku služby ‘strtclk‘ uzavřít do kritické sekce (chráněné zákazem přerušení).
- Služba srtrclk nejprve ověří, jsou-ti vůbec hodiny pozastaveny (test `defclk == 0`), a pokud ano, sníží počet pozastavení dekrementováním proměnné `defclk`. Je-li pak výsledná hodnota proměnné nulová, je zapotřebí hodiny opět rozběhnout.
- V takovém případě služba zjistí počet odložených zpracování obslužné rutiny časovače, který je uložen v globální proměnné `clkdiff`, a tuto proměnnou vynuluje (aby se do ní při příštím pozastavení hodin opět ukládaly korektní hodnoty). Zároveň služba odpovídajícím způsobem sníží hodnotu globální proměnné `preempt`. Operační systém v ní udržuje informaci o tom, kolik času zbývá aktivnímu procesu do přeplánování v rámci sdílení času. Musíme ji tedy samozřejmě snížit právě o tolik, kolikrát obslužná rutina časovače ‘nebyla k dispozici‘.

- Funkce použije globální proměnnou `slnempty`, aby zjistila, je-li delta List neprázdný. Čekají-ti nějaké procesy na ‘probuzení‘, postupuje služba následujícím způsobem:
  - Všem procesům, jejichž zbývající čas je nižší než počet ‘odložených‘ zpracování obslužné rutiny přerušení časovače, nastaví ‘deltu‘ na nulu. Navíc je při každém kroku nutné snížit obsah proměnné `difftime`, protože vynulováním ‘delty‘ vlastně ‘jakoby‘ patřičný počet odložených přerušení obsloužíme.
  - Jakmile narazíme na proces s příliš velkou ‘deltou‘, přestaneme frontu procházet (to zajisti příkaz `else break`). Příliš velká ‘delta‘ totiž znamená, že takový proces bude čekat ještě déle.
  - ‘Deltu‘ takového procesu však ještě musíme snížit o zbývající počet odložených přerušení, který máme v proměnné `difftime`.
  - Nakonec stačí zavolat službu ‘wakeup‘, která automaticky zajistí převedení všech procesů s nulovou ‘deltou‘ do ready fronty a korektně nastaví globální proměnné.

```
/* clk_iohandler.c */

void interrupt clk_iohandler()
{
     static cnt = 1;

     if (--cnt == 0) {
          cnt = CLKDIV;
          if (defclk) {
               clkdiff++;
               return;
          }
          if (slnempty  &&  --*sltop == 0)
               wakeup();
          if (--preempt == 0)
               resched();
     }
}
```

- První příkaz ‘if slouží k dělení hodinové frekvence - jeho tělo bude provedeno pouze při každém CLKDIV-tém přerušení.
- Obslužná rutina pak nejprve ověří, nejsou-li hodiny pozastaveny; pokud by tomu tak bylo, inkrementuje pouze proměnnou `clkdiff` a ihned skončí.
- Pokud hodiny pozastaveny nejsou, ověří rutina, je-li delta list neprázdný a případně dekrementuje ‘deltu‘ jeho prvního procesu (na tu ukazuje globální proměnná `sltop`).
- Je-li ‘delta‘ po dekrementaci nulová, zavolá se služba ‘wakeup‘ pro ‘probuzení‘ všech procesů, které to vyžadují.
- Nakonec rutina sníží zbývající čas aktivního procesu uložený v proměnné `preempt`, a je-li to zapotřebí, zajistí přeplánování.

Preempce a bezpečnost

- Může uživatelský program OS zabránit, aby mu odebral procesor a tedy narušit práci OS? Existují dvě možnosti:
  - Uživatelský proces by si extrémně zvýšil prioritu. Pak by byl neustále první v ready frontě (pokud by byl přeplánován) a neustále by měl přidělen procesor.
  - Uživatelský proces by zakázal přerušení a tím by vyřadil z činnosti systémový časovač

Zvýšení priority

- Proces vůbec neví, kam systém ukládá jeho prioritu. Pro změnu priority proto musí nutně volat službu operačního systému; je velmi snadné tuto službu naprogramovat tak, aby si proces nemohl přidělit prioritu příliš vysokou.
- V systému bez ochrany paměti by se proces samozřejmě mohl pokusit vyhledat tabulku procesů a svou prioritu zvýšit přímo; nemáme-li však k dispozici ochranu paměti existuje tolik nejrůznějších způsobů, jak omylem nebo úmyslně zapříčinit zhroucení operačního systému, že tato možnost vlastně už nic nezhorší.

Zakázání přerušení

- Nebezpečnější je případ, kdy proces zakáže přerušení. Řešení je jednoduché, závisí ale na technických prostředcích: uživatelským procesům je prostě nutné zamezit zakazování přerušení.
- Moderní procesory, které rozlišují **uživatelský** a systémový **chráněný** režim, tuto problematiku řeší prostě tak, že v uživatelském režimu zakazovat přerušení nemůžeme (instrukce zákazu přerušení je tzv. privilegovaná – lze provést pouze na určité úrovni oprávnění) - pokus o tuto akci vyvolá výjimku, kterou obslouží operační systém a s ‘provinilým‘ procesem něco vhodného provede - nejspíše jej okamžitě ukončí.

Ovladače a bezpečnost

- Při návrhu systému ovladačů je zapotřebí věnovat zvláštní pozornost také **bezpečnosti** celého operačního systému. Ovladače totiž musí mít přístup k zařízením, která ovládají, na té nejnižší úrovni; to automaticky znamená, že alespoň část kódu každého ovladače musí pracovat v systémovém režimu práce procesoru, kdy je '**všechno dovoleno**', a mohla by tedy potenciálně zapříčinit zhrouceni systému.
- Neexistuje samozřejmě obrana proti chybě v kódu ovladače nebo dokonce proti úmyslně destruktivně napsanému ovladači. Ovladače zařízení jsou **součástí** operačního systému a celý systém je pochopitelně jen tak spolehlivý, jak je spolehlivý jeho nejslabší článek (proto např. 32-bitové Windows vyžadují digitálně podepsané ovladače) .
- Kromě toho je však zapotřebí věnovat pozornost také tomu, aby prostřednictvím i bezchybně napsaného ovladače neměl možnost vyvolat omylem nebo úmyslně nějakou destruktivní činnost obyčejný uživatelský program.

Příčiny nebezpečí

- Systém ovladačů zařízení patří mezi nejrizikovější části operačního systému z hlediska bezpečnosti vůbec.
- Prvním problémem, na který naráží zabezpečení ovladačů zařízení, je **požadavek na efektivitu a rychlost**. Programátoři pak často zrychlují jednotlivé služby velmi jednoduchým způsobem - odstraní z nich všechno, co zdržuje; v první řadě zabezpečení.
- Je samozřejmé, že zabezpečený systém bude o něco pomalejší než systém který se o bezpečnost vůbec nijak nestará. Kvůli zabezpečení je zapotřebí provádět množství testů, kontrolovat vstupní parametry funkcí a podobně; to pochopitelně nějakou dobu trvá.
- S výjimkou zcela uzavřených operačních systémů se však vyplatí bezpečnosti se věnovat - jinak se ušetřené milisekundy po prvním výpadku změní ve ztracené týdny a měsíce, ne-li roky práce. Mírné riziko, uzavřené kompletně uvnitř operačního systému a nepřístupné uživatelským programům, může být přípustné, jestliže přinese výrazné zrychlení; na každé takové riziko by však mělo být v dokumentaci systému velmi výslovné upozornění.

- Větším nebezpečím pro spolehlivost systému ovladačů je jejich komplikovanost. Porovnejme např. situaci při práci ovladače se situací při přístupu k operační paměti:
  - K operační paměti přistupuje jediný proces, fyzickými účastníky celé operace jsou pouze procesor a paměť sama.
  - Při práci se zařízením je situace daleko složitější: samo zařízení (např. řadič diskety) komunikuje s médiem (s konkrétní disketou), přitom používá paměťové buffery a komunikuje s procesem. Fyzickými účastníky celé operace jsou tedy zařízení, médium, procesor a operační paměť.
- Systém zabezpečení musí zajistit korektní vzájemnou spolupráci všech čtyř účastníků přenosu a každého z nich navíc musí chránit před nežádoucím přístupem.

Nebezpečné příkazy

- Jedním z potenciálních nebezpečí jsou příkazy, které by mohly vést k chybě nebo přímo k destrukci zařízení. Ovladač musí provedení takových příkazů zamezit nebo je musí modifikovat takovým způsobem, aby bylo nebezpečí odstraněno.
- Programátorský folklór např. uvádí, že vhodným časováním nastavování hlaviček pevného disku je možné dosáhnout rezonance, hlavičky rozkmitat a disk zničit. Pokud by tomu tak totiž skutečné bylo, musí ovladač pevného disku mimo jiné kontrolovat seřazení požadavků ve frontě i z hlediska této rezonance, a pokud by k ní mohlo dojít, musí pořadí požadavků přeuspořádat bez ohledu na efektivitu (přesněji řečeno s ohledy na efektivitu, ale pouze v rámci uspořádání. která rezonanci vyvolat nemohou).
- Naštěstí u moderních řadiče disků už toto nebezpečí nehrozí, řadiče si samy určují pořadí zpracování příkazů.

„Callback“ rutiny

- Některé ovladače využívají tzv. „callback“ rutiny. To jsou funkce připravené uživatelem; adresa takové funkce je předána ovladači a ten pak funkci zavolá při případné aktivitě zařízení (např. ošetření chyby, apod.)
- „Callback“ rutiny jsou potenciálně **velmi nebezpečné**, protože uživatelský kód je zde vlastně prováděn jako součást ovladače zařízení. V principu je sice možné vytvořit bezpečné konvence pro používání „callback“ rutin; daleko lepší je však kompletně se „callback“ rutinám vyhnout a namísto nich pro informování uživatelského programu o aktivitě zařízení použít mechanismus zpráv nebo semaforů.

Přístup k paměti

- Velkým bezpečnostním hazardem je přístup k operační paměti. Uvědomme si, že ovladač - který pracuje v systémovém režimu procesoru - má přístup k celé operační paměti. Uživatelské rutiny pro vstup přitom ovladači předávají adresy bufferů, do nichž má ovladač uložit načtená data.
- Pokud návrhář systému ovladačů nebude mít tento faktor na paměti a nezabezpečí, aby předávané buffery jistě patřily procesu, který službu vyvolal, mohl by kterýkoli proces velmi snadno dosáhnout zhroucení systému: stačilo by vyžádat si třeba čtení sektoru z disku a jako adresu bufferu uvést adresu někde uvnitř chráněných dat operačního systému (pokud proces nezná adresy dat operačního systému, stačí úplně generovat takové adresy náhodně - dříve nebo později se jistě ‘strefí’).
- Jestliže je počítač vybaven virtuální pamětí, musí ovladač - který samozřejmě dostane od programu logickou adresu bufferu - adresu sám přeložit na fyzickou, protože téměř všechna zařízení pracují přímo s fyzickými adresami v operační paměti. To ale samozřejmě přináší mnoho rizik, která musí programátor mít na paměti a odstranit - ovladač musí úzce spolupracovat se správcem paměti, který samozřejmě stránku nebo stránky odpovídající bufferu nesmí přidělit nikomu jinému, dokud není přenos ukončen.

Změna média

- Dalším rizikem z hlediska bezpečnosti je možnost výměny média. Představme si, že operační systém ukládá nějaké údaje na disketu; nejprve zapíše vlastní data a potom chce ještě zaznamenat provedené změny v adresáři diskety. Mezi zápisem dat a záznamem do adresáře je však odpovídající proces na čas pozastaven (mohl se třeba objevit jiný proces s vyšší prioritou); uživatel mezitím disketu vyndá a nahradí jinou. Výsledkem bude nekonzistentní obsah obou disket a pravděpodobná ztráta dat.
- Existují v zásadě dvě cesty jak se může ovladač tomuto riziku ubránit:

- První z nich používá většina rozumných počítačů a pracovních stanic při práci s disketou: disketu není možné vyjmout jinak, než příkazem programu. Operační systém tedy v každém okamžiku ‘ví’, jestli mohlo dojít k výměně média - protože pokud sám operační systém médium nevysunul, uživatel to udělat nemohl. (bohužel u klasických disketových mechanik 3,5“ to možné není, tlačítko na vysunutí diskety je mechanické. U DVD je situace jiná, tlačítko je elektronické a disk vysunuje operační systém).
- Nutnost využít některé ze služeb systému pro vyjmutí média může být někdy nepohodlná; pak je možné ji kompenzovat tím, že zařízení vyjmutí média detekuje a operačnímu systému je ohlásí.
- V obou případech však musí mít operační systém možnost po opětovném zasunutí média ověřit, jedná-li se o původní médium, na kterém jsou rozpracovaná nekonzistentní data, nebo ne. V ideálním případě by mělo být každé médium vybaveno vlastním sériovým číslem, které se liší pro každá dvě média, a systém jej může snadno ověřit; Pokud medium není vybaveno seriovým číslem, pak je nutné kontrolovat, nezměnil-li se obsah media; je-li takový test dostatečně komplexní, je dost spolehlivý, ale bohužel také dost pomalý.

Výpadek systému

- Vyhrocením problémů popsaných v minulém odstavci je možnost výpadku systému (např. vinou přerušení dodávky elektrické energie nebo poruchy některé části technického vybavení). Má-li být systém opravdu bezpečný. nemělo by k výpadku v žádném případě dojít ve chvíli, kdy jsou data na jakémkoli médiu nekonzistentní.
- Toto je snad jediná oblast, ve které je na tom poměrně špatně jinak špičkový operační systém - totiž UNIX. Běžné implementace UNIXu totiž udržují poměrně rozsáhlé oblasti dat, která by měla být na disku, v operační paměti, a na disk je zapisují až v případě potřeby. Výpadek systému pak samozřejmě může zanechat disky ve stavu poněkud rozháraném (dnes jsou na tom stejně i Windows).
- Existují v podstatě tři způsoby obrany proti těmto problémům; první, přístupný každému operačnímu systému, druhý, náročný na technické vybavení a třetí, náročný na kapacitu vnější paměti. Jen třetí mechanismus však může zajistit skutečnou bezpečnost operačního systému:

1. Nejjednodušší technikou je vytvářet ovladače tak, aby data na kterémkoli zařízení byla v nekonzistentním stavu vždy co možná nejkratší dobu. Tím se snižuje pravděpodobnost chyby, protože i když k výpadku systému dojde, máme poměrně vysokou šancí, že všechna data jsou konzistentní.
2. Druhý mechanismus vyžaduje zálohovaný napájecí zdroj, který v případě výpadku proudu okamžitě vyvolá přerušení, které aktivuje ‘nouzovou‘ rutinu operačního systému. Ta okamžitě uvede všechna data do konzistentního stavu a uloží na vnější paměti kompletní stav systému, včetně času, ve kterém k výpadku došlo. Zálohovaný zdroj musí být samozřejmě schopen dodávat energii dost dlouho na provedení všech těchto akcí. Při nejbližším startu systém detekuje k čemu došlo, a obnoví původní stav všech údajů. Tento mechanismus dokáže zcela zabránit všem problémům, které mohou být důsledkem výpadku elektrického proudu; není však nic platný pro případ chyby samotného technického vybavení počítače.
3. Skutečně bezpečný systém proto automaticky vytváří a ukládá kopie svého stavu preventivně, buď v určitých časových intervalech, nebo před významnými změnami (nebo obojí) – tzv. **transakční systém**. Dojde-li potom skutečně k výpadku, je možné obnovit stav systému před poslední změnou a zopakovat příkazy, které byly po této změně provedeny; to sice znamená jistou ztrátu času, zaručeně však zanedbatelnou ve srovnání s případnou ztrátou kompletních zpracovávaných dat. Poznamenejme. že tento velmi luxusní, ale i náročný zabezpečovací systém není žádnou utopií - jsou jím vybaveny např. snad všechny moderní databázové servery.

Systém souborů

- Vznik **systému souborů** vynutila potřeba ukládat na jediný disk množství ‘balíků‘ na sobě navzájem nezávislých údajů a tedy potřeba zavést nad diskem nějakou uživatelsky pohodlnou a strojově nezávislou logickou strukturu.
- Z jiné strany se na systém souborů můžeme dívat také jako na nejvyšší fázi virtualizace disku.
- Obecnou tendencí dnešních systémů je posilování logické struktury systému souborů a odbourávání posledních vazeb na konkrétní fyzickou strukturu disku. Velmi často se o soubory stará samostatný server; běžně s ním však programy nekomunikují přímo, ale prostřednictvím ovladačů logických zařízení, odpovídajících souborům. To do značné míry zvyšuje **flexibilitu** operačního systému - programy nemusí rozlišovat přístup k souborům a přístup ke skutečným zařízením, všechny rozdíly skrytě obslouží operační systému.

- Některé operační systémy volí právě opačný přístup, na první pohled o něco komplikovanější, ale v praxi daleko výhodnější: systém souborů je natolik flexibilní, že zahrnuje i všechna zařízení a programátor tedy s kterýmkoli zařízením může pracoval jako se souborem.
- Z hlediska vlastních operací nad souborem nebo zařízením je to naprosto jedno; výhoda je v tom, že pro zařízení můžeme použít rozsáhlý aparát vlastnických práv a řízeného sdílení, který bývá součástí systému souborů. Dobrým příkladem takového operačního sytému je právě UNIX.

Implementace souborového systému

- Pro implementaci systému souborů musíme obvykle rozdělit disk na:
  - **alokační jednotky**, kde budou umístěna data
  - **Informace o řetězení alokačních jednotek** - informace, které určují přidělení alokačních jednotek jednotlivým souborům (která alokační jednotka následuje).
  - **index** - ten obsahuje o každém souboru základní informace, jako je typ souboru, délka, doba vytvoření, doba poslední modifikace, přístupová práva uživatelů a podobně. V indexu by mohlo být uloženo i jméno souboru a u jednodušších systémů tomu tak skutečně bývá; daleko výhodnější však je po vzoru UNIXu jména souborů umístit jinde
- Logická struktura systému souborů je pak poměrně jednoduchá. Každý soubor je jednoznačně určen položkou v indexu; ta obsahuje všechny důležité informace o souboru a také číslo první alokační jednotky, ve které jsou data souboru. Jestliže chceme mít přístup k pokračování souboru, zjistíme z informací o řetězení, ve které alokační jednotce soubor pokračuje. Tak můžeme postupovat až ke konci souboru.

Konkrétních implementací může být celá řada:

- Alokační jednotkou může být v nejjednodušším případě sektor (512 bytů). Na velkém disku je však v takovém případě alokačních jednotek příliš mnoho - řetězení tolika alokačních jednotek potom zabírá na disku ‘zbytečně‘ mnoho místa. Nejčastěji je proto alokační jednotka složena ze čtyř nebo osmi sektorů (obyčejně mocnin 2).
- Nejjednodušší implementace pro řetězení alokačních jednotek je alokační tabulka (takzvaná **FAT** – **File Allocation Table**). To je tabulka čísel, která má právě tolik položek, kolik je na disku alokačních jednotek. Chceme-li pak zjistit, která alokační jednotka logicky následuje za jednotkou číslo ‘n‘, podíváme se prostě do ‘n‘té položky alokační tabulky.

File Allocation Table

Alokační jednotka	Následující alokační jednotka	
0	neobsazena	
1	3	
2	135	
3	-1	Další alokační jednotka už není, 3 je poslední
.	.	
.	.	
.	.	
N-3	0	Volná alokační jednotka
N-2	0	Volná alokační jednotka
N-1	0	Volná alokační jednotka

- Bezpečnější, ale programátorsky méně pohodlná, metoda ukládá řetězící informace přímo do alokačních jednotek (např. implementace NTFS a souborové systémy v UNIXu). Takový disk je poměrně velmi dohře chráněn proti zničení dat - i v případě, že je poškozena část disku, lze údaje ze zbytku obnovit (podobný mechanismus, který používá správce paměti). Zničíme-li však alokační tabulku FAT, jsou data na disku definitivně ztracena (resp. nepoznáme, která alokační jednotka následuje).
- **Index** bývá obvykle implementován nejjednodušším možným způsobem jako tabulka pevné velikosti, jejíž položky odpovídají jednotlivým souborům. Na úrovni takovéhoto systému souborů není samozřejmě soubor identifikován svým jménem, ale číslem, které udává jeho pozici v indexu. Pro identifikaci souborů podle jmen slouží vyšší vrstva systému souborů, které říkáme **systém adresářů.**

Systém adresářů

- **Systém adresářů** sám využívá systému souborů a umožňuje uživateli, aby k souborům přistupoval prostřednictvím logických jmen. Princip systému adresářů spočívá v tom, že ne každý soubor musí obsahovat uživatelská data; některé soubory mohou obsahovat také **seznam jmen** spojených s **čísly souborů**. Jeden soubor (třeba ten první v indexu, který má číslo 0), má pak speciální postavení tzv. **kořenového adresáře** - zadá-li uživatel nějaké jméno, začne jej systém adresářů vyhledávat právě v tomto souboru.
- Popsaný mechanismus nabízí uživateli daleko větší flexibilitu, než kdybychom jména souborů ukládali přímo do indexu. Bez jakýchkoli dodatečných prostředků máme k dispozici **hierarchickou strukturu adresářů**: jméno v kořenovém adresáři může určovat další adresář, ve kterém se bude vyhledávat další jméno přesně stejným způsobem, jakým se první jméno hledalo v adresáři kořenovém.
- Navíc jediný soubor může být určen řadou nejrůznějších jmen (tzv. **link**), dokonce v různých úrovních adresářů. To je velmi výhodné zvláště ve víceuživatelském prostředí. Struktura adresářů ani nemusí být přísně hierarchická - kterýkoli adresář může obsahovat u nějakého jména třeba odkaz na adresář kořenový .

Formátované soubory

- Základní systém souborů nabízí programům velmi jednoduchý pohled na soubory: soubor je prostě řetězec bytů určité délky.
- Existuje samozřejmě řada problémů, pro které je tento přístup výhodný; daleko více úloh však bezpochyby využívá soubory formátované.
- Rozumný operační systém by proto měl podporovat kromě výše uvedeného neformátovaného souboru alespoň dva nejčastěji používané formáty:
  - textový soubor organizovaný po řádcích (s identifikací řádku pomocí znaku CR-LF, LF, …)
  - databázový soubor s pevnou délkou záznamu
- Kvalitní systémy by navíc měly podporovat i náročnější, ale zato daleko praktičtější databázový formát s proměnnou délkou záznamu;

NTFS

- NTFS je velmi starý systém. Počítače dost dlouho pracovaly na systému DOS s FAT. NTFS byla vytvořena pro operační systém Windows NT 3.1.
- Jedná se o souborový systém Microsoftu podporovaný jejich operačními systémy od Windows NT 3.1 po současné operační systémy Windows. Jinými operačními systémy není s plnou funkčností podporován. Existují tzv. doplňky, které si můžete dokoupit od nezávislých zdrojů např. do Linuxu, Windows 95 aj., nedoporučuje se to ale pro normální činnost – pouze příležitostně přečíst data.

Fyzická struktura

Oddíl (partition) NTFS může mít velikost 4 TB.

- **Struktura oddílu**
  - Stejně jako každý systém, NTFS dělí využitelné místo na alokační jednotky - bloky bytů použité pohromadě. NTFS podporuje skoro všechny velikosti alokačních jednotek - od 512 B až do 64 kB. Alokační jednotka velikosti 4096 bytů je považována jako standard.
  - Disk NTFS je symbolicky rozdělen do dvou částí. Prvních 12 % disku je označena jako tzv. **oblast MFT** - místo, ve kterém se rozrůstá metasoubor **MFT (Master File Table)**. Zapisovat data do tohoto místa není možné. Oblast MFT je vždy zachována prázdná, aby nejdůležitější servisní soubor (MFT) nebyl při růstu fragmentován. Zbývajících 88 % disku již představuje použitelné místo pro ukládání dat.
  - Ačkoli část pro soubory obsahuje veškeré fyzicky volné místo, je v něm také začleněna část prostoru MFT. Mechanismus používání MFT je následující: když soubory už není možno zapsat do použitelného prostoru, oblast MFT se jednoduše zredukuje, aby uvolnila místo pro zapisované soubory. Při uvolnění použitelného místa se MFT může zase rozšířit. Takže je možné pro běžné soubory, aby zůstaly v tomto prostoru a je to normální. Systém zkouší zachovat tuto část volnou, avšak občas selhává. Přesto metasoubor MFT může být fragmentovaný, i když to třeba není nežádoucí.

MFT a jeho struktura

- Souborový systém NTFS je úžasný svým výkonným složením. **Každá součást je soubor - dokonce i systémové informace.** Nejdůležitějším souborem NTFS je soubor nazvaný **MFT - Master File Table** (hlavní tabulka souborů). Je uložen v oblasti MFT a představuje centralizovanou složku všech zbývající souborů včetně sebe.
- MFT je rozdělen na záznamy fixní délky (obvykle 1 kB) a každý z nich koresponduje s nějakým souborem. Prvních 16 souborů je určeno pro vnitřní potřebu systému a nejsou přístupné operačním systémem. Jsou nazvány jako metasoubory a první je MFT sám. Těchto 16 elementů MFT představuje jedinou část, která má fixní umístění na disku. Zajímavostí je, že kopie prvních 3 záznamů je kvůli spolehlivosti (jsou velice důležité) uložena do středu disku. Zbytek MFT může být uložen kdekoliv jako ostatní soubory. Je možné obnovit jeho pozici za pomocí sebe sama s použitím jeho základní polohy - prvního MFT elementu.

Metasoubory

Prvních 16 souborů (metasouborů) jsou systémové a každý z nich je odpovědný z hlediska systémových operací. Výhoda rozdělení na takové moduly spočívá v úžasné flexibilitě - např. u FAT je fyzická porucha v oblasti FAT osudová pro všechny diskové operace. Pokud jde o NTFS, pro zamezení jakékoliv havárie povrchu systém může přemístit nebo fragmentovat všechny systémové oblasti až na prvních 16 elementů MFT. Metasoubory jsou v rootu NTFS, jejich název začíná znakem "$", takže je obtížné pomocí běžných prostředků získat o nich nějaké informace. Kupodivu je u těchto souborů uložena informace o velikosti a je tedy možné při pohledu např. na velikost $MFT zjistit, kolik zabírá katalogizace celého disku.

$MFT	Sám MFT
$MFTmirr	Kopie prvních 16 záznamů MFT umístěná do středu disku
$LogFile	soubor pro protokolování
$Volume	interní informace - název oddílu, verze systému souborů, atd.
$AttrDef	soupis standardních atributů souborů na oddílu
$.	složka rootu
$Bitmap	bitová mapa volného místa oddílu
$Boot	boot sektor (bootovatelný oddíl)
$Quota	soubor, ve kterém jsou uložena práva uživatele na užití místa disku (od NT5)
$Upcase	Soubor - tabulka shody malých a velkých písmen v názvech souborů na daném oddílu. Je důležitá, jelikož názvy soborů NTFS jsou zapsány v Unicodu, který tvoří 65 tis. různých znaků a není jednoduché vyhledávat kvůli ekvivalenci malých a velkých písmen.

Soubory a toky (streams)

- První nucenou částí je záznam v MFT. V MFT jsou uvedeny všechny soubory disku. Všechny informace o souboru až na samotná data souboru jsou zde uložena: název souboru, jeho velikost, pozice fragmentů na disku atd. Pokud jeden záznam v MFT na informace nestačí, pak je použito několik záznamů nezávisle jeden na druhém. Nepovinnou součástí jsou toky dat souborů.
  - soubor neobsahuje žádná data a potom se pro něj nepoužije volné místo.
  - soubor není příliš velký a potom se učiní velice zdařilé rozhodnutí: data souboru jsou uložena právě v MFT v místě mimo hlavní data z omezením na jeden záznam MFT. Soubor o velikosti stovek bytů zpravidla nemá fyzický obraz v základní souborové oblasti. Všechna taková data souborů jsou uložena právě na jednom místě - v MFT.
- Je zde jeden zajímavý případ s daty souboru. Každý soubor na NTFS má dost abstraktní pojetí - nemá žádná data, má jen toky. Jeden z toků má pro nás zcela obvyklý smysl - data soboru. Ale většina atributů souboru jsou také toky! Takže nemáme nic než základní příznak souboru, kterým je jen **číslo v MFT** a zbytek je nepovinný.
- Tato abstrakce může být velice vhodně využita - např. můžeme uložit k souboru ještě další tok, i když v něm máme nějaká data uložena, třeba informaci o autorovi a obsahu souboru. Je zajímavé, že tyto dodatečné toky nejsou viditelné s použitím běžných prostředků: viditelná velikost souboru je jen velikost hlavního toku obsahujícího tradiční data. Takže může nastat třeba situace, že smazáním souboru s nulovou délkou se uvolní 1 GB, protože některý program využívající přídavných toků si do nich uložil informaci o velikosti 1 GB.
- Jméno souboru může sestávat z jakýchkoli znaků včetně celé národní abecedy, protože data jsou zastoupena sadou Unicode - 16bitovým podáním umožňujícím 65535 různým znaků. Maximální délka názvu souboru je 255 znaků.

Složky (folders)

- Složka na NTFS je specifický soubor nesoucí odkazy na ostatní soubory a složky zakládající se na hierarchické struktuře dat disku. Soubor složky je rozdělen na bloky, kde každý z nich obsahuje název souboru, základní atributy a odkaz na element MFT, který již podává kompletní informace o elementu složky.
- Vnitřní struktura složky je binární strom. To znamená, že při hledání souboru s daným jménem v lineární složce jako tomu je např. u FAT, by operační systém musel projít přes všechny elementy složky než najde ten správný.
- Binární strom alokuje názvy souborů takovým způsobem, aby bylo vyhledání souboru rychlejší - tj. pomocí obdržených binárních odpovědí na dotaz o pozici souboru. Binární strom (tree) je schopen dát odpověď na otázku, ve které skupině je požadovaný název situován - nad či pod daným elementem.
- Začínáme s dotazem na element ve středu a každá další odpověď zužuje prohledávané pole dvakrát (analogická je nám dobře známá numerická metoda integrování půlením intervalu). Soubory jsou tříděny abecedně a odpověď na dotaz je splněna nejběžnějším způsobem - srovnáním počátečních písmen. Prohledávaná oblast, která byla dvakrát zúžena, se znovu prohledává stejným způsobem znovu od středního elementu.
- Je nutné se uvědomit, že abychom našli jeden soubor mezi 1000 soubory, např. u FAT by se muselo provést kolem 500 srovnání (z teorie pravděpodobnosti se soubor nejpravděpodobněji nachází uprostřed) a u systému založeném na stromu něco kolem 10. Šetření časem je tedy viditelné.
- Pro jednoduchou navigaci na disku není nutné jít kvůli každému souboru do MFT, je jen nutné přečíst nejběžnější informace o souborech ze souborů složek. Hlavní složka disku - root - se vůbec ničím neliší od ostatních složek až na speciální odkaz na ni na začátku metasouboru MFT.

Protokolování

- NTFS je systém zajištěný proti chybám, který dokáže sám opravit prakticky jakoukoli reálnou poruchu. Kterýkoliv moderní souborový systém je založen na koncepci transakcí - akce provedená zcela a správně nebo vůbec neprovedena. NTFS nemá střední (chybné nebo nesprávné) podmínky - kolísání množství dat nemůže být děleno na před a po poruše způsobené nějakou chybou - je buď hotové nebo stornované.
  - **Př. 1.** Provádí se zápis dat na disk. Náhle se zjistí, že je nemožné zapsat do tohoto místa, kam jsme se rozhodli zapsat další část dat, protože je zde povrch poškozen. Chování NTFS je pro tento případ logické: transakce zápisu je zcela odsunuta - systém si uvědomí, že zápis do tohoto místa není efektivní. Místo je označeno jako chybné a data se ukládají do jiného místa a je spuštěna nová transakce.
  - **Př. 2.** Případ je komplexnější - provádí se zápis dat na disk. Náhle vypadne napájení a systém se restartuje. Ve které fázi se zápis zastavil a kde jsou data? Zde nám přichází na pomoc deník transakcí. Systém si uvědomí žádost o zápis na disk a zaznamená do metasouboru $LogFile tuto okolnost. Po opětovném nastartování systém soubor prozkoumá, aby zjistil neukončenou transakci, která byla přerušena zhroucením a výsledek je tedy nepředvídatelný. Všechny tyto transakce se stornují: místo, kde se prováděl zápis se znovu označí jako volné a systém zůstává stálý v celku. A co situace, když chyba nastane při zápisu do protokolu? To není vůbec osudné: transakce se buď ještě nespustila (je zde jen pokus o zápis záměru ji vytvořit) nebo už byla kompletní - pak je zde jen pokus o zápis, že transakce byla splněna. Při následujícím zavedení systému si sám systém zcela ujistí, že vše je zapsáno správně a nebude upozorňovat na neukončenou transakci.

- Protokolování není absolutní lék na vše ale jen prostředek k redukování počtu chyb a systémových poruch. Zkušenost ukazuje, že NTFS je obnovena za úplně korektních okolností dokonce i ve chvíli velmi velkého zatížení diskovou aktivitou. Můžete klidně defragmentovat disk a ve špičce procesu stlačit reset - pravděpodobnost ztráty dat dokonce v tomto případě bude velmi malá.
- **Je důležité si uvědomit, že ale systém obnovy NTFS garantuje jen správnost celého systému souborů, nikoli vašich dat**. Pokud provádíte zápis na disk a došlo k zhroucení - správná data často **nemůžou být obnovena**.

Komprese

- Soubory na oddílech NTFS mají jeden užitečný atribut - komprese. NTFS je postavena s podporou diskové komprese. Dříve pro to byly použity utility Stacker nebo DoubleSpace aj. Jakýkoliv soubor či složka můžou být jednotlivě uloženy na disk v komprimované formě a tento proces nijak neomezuje aplikace. Komprese souborů má velmi vysokou rychlost.

Zabezpečení

- NTFS skýtá mnoho prostředků k rozlišení práv objektů. Práva sytému souborů NTFS jsou úzce spojené se samotným systémem a to znamená, že nejsou povinné při fyzickém přístupu na disk jiným operačním systémem.

- **Pevné odkazy (hard links)**

  Pevný odkaz vzniká, když stejný soubor má dva názvy (některé záznamy složky ukazují na stejný záznam v MFT). Připusťme, že stejný soubor má názvy 1.txt a 2.txt: pokud uživatel smaže soubor 1, soubor 2 zůstane. Pokud smaže soubor 2, soubor 1 zůstane. To znamená, že oba názvy jsou úplně totožné od okamžiku vytvoření. Soubor je fyzicky smazán pouze, pokud je jeho poslední jméno smazáno.
- **Symbolické odkazy (symbolic links)**

  Zde je mnohem více praktických možností umožňujících vytvořit virtuální složky. Aplikace jsou dostatečně široké: nejprve je to zjednodušení systému složek. Když se vám nelíbí název složky Documents and settings\Administrator\Documents, můžete na něj vytvořit odkaz do rootu a systém bude komunikovat se složkou přes původní cestu, ale vy - s mnohem kratším názvem úplně ekvivalentním k ní. Odkaz se dá zrušit příkazem *rm*. Při smazání odkazu Průzkumníkem čí jiným programem, se však smaže i obsah toho, na co odkaz ukazuje!
- **Šifrování**

  Každý soubor složky může být zašifrován a pak není možné, aby ho někdo přečetl jinou instalací Windows. V kombinaci se standardní a velice bezpečným heslem do samotného systému tato možnost nabízí bezpečnost vámi vybraných důležitých dat pro většinu aplikací.
- **Kvóty**

  Další vlastností je kvótování. Administrátor může každému uživateli nastavit, kolik může do dané složky uložit dat. Výhodné pro servery nebo stanice s přístupem více lidí. Toto vlastnost se používá stejně jako serverů Novell Netware.

Typy svazků (volumes) NTFS

- **Jednoduché svazky (simple volumes)**
  Jde o svazky vytvořené na jednom fyzickém disku přes jeden nebo libovolný počet oddílů NTFS, které spojíme dohromady.
- **Rozložené svazky (spanned volumes)**
  Pokud jednoduchý svazek rozšíříme přes několik fyzických disků, stává se jednoduchý svazek rozloženým svazkem. U dynamických disků je možné připojit do již existujícího svazku další disk a rozšířit tak místo kdykoliv.
- **Prokládané svazky (striped volumes)**
  Data takového svazku jsou ukládána střídavě na dva nebo více fyzických disků. Data těchto svazků jsou alokována střídavě a rovnoměrně na disky prokládaného svazku. Tyto svazky značně urychlují přístup k harddisku.
- **Zrcadlené svazky (mirrored volumes)**
  Jde o svazky vylučující chyby, které ukládají data ve dvou kopiích na dva fyzické disky. Poskytují zálohu dat tím, že používají kopii (zrcadlení) svazku, aby duplikovali informace uložené na svazcích. Zrcadlo je uloženo na jiný fyzický disk. Pokud data na jednom disku začnou vykazovat chyby a stanou se nepoužitelnými, systém pokračuje v operaci a používá nepoškozené data druhého disku. Zrcadlený svazek je pomalejší než svazek RAID-5 při operacích čtení, ale je rychlejší při operacích zápisu.
- **Svazky typu RAID-5**
  Jde o svazky, které k vyloučení chyb dat a parity prokládají obsah přes tři nebo více fyzických disků. Když zhavaruje část fyzického disku, je možné znovu vytvořit data, která byla na zhavarované části ze zbývajících dat a parity. Svazky RAID-5 jsou dobrým řešením pro prostředí s počítači, kde je vysoká aktivita čtení dat.

Sdílení souborů

- Důležitou součástí systému souborů je i mechanismus **sdílení**. Není možné nechat zcela nekoordinovaně dva procesy měnit údaje v jediném souboru. Na druhé straně, vyjma nejjednodušších systémů (mezi které patří např. XINU), si nemůžeme dovolit zakázat dvěma procesům současný přístup k jedinému souboru - nejmarkantnějším příkladem jsou databázově systémy, kde bývají často databáze sdíleny řadou nejrůznějších programů, které je doplňují i zpracovávají.
- Obvykle operační systém umožňuje procesům zvolit **způsob přístupu** k souboru. Zvolí-li proces tzv.. **sdílený přístup,** není mu umožněno obsah souboru měnit, zato však může k témuž souboru přistupovat libovolné množství procesů najednou.
- Další možností je samozřejmě **výhradní přístup**, při kterém má proces nad souborem plnou kontrolu, ovšem za tu cenu, že žádný jiný proces nemůže se souborem pracovat.

- Moderní operační systémy, které umožňují práci s formátovanými soubory, mohou využívat i lepší metodu současného přístupu k souborům. Při ní každý proces otevírá soubor ‘sdíleným‘ způsobem a teprve ve chvíli, kdy chce nějakou část souboru měnit, vyžádá si od systému **dočasné vyhrazení** této části souboru - nejčastěji se může jednat o jeden databázový záznam. Operační systém záznam ‘**zamkne**‘ a umožní procesu jej měnit; přitom však jiné procesy mohou pohodlně zpracovávat ostatní části souboru (jiný proces může dokonce měnit jinou část souboru ve stejné chvíli, kdy první proces mění ‘svůj‘ záznam).
- Pro implementaci takovéhoto ‘inteligentního‘ systému souborů je obvykle nejvýhodnější vytvořit opět **server**, tj. samostatný proces. Ten má jako jediný přístup k souborům; všechny ostatní procesy mohou se soubory pracovat pouze prostřednictvím serveru (požadavky serveru obvykle předávají prostřednictvím systému zpráv). Server tak může poměrně velmi snadno koordinovat požadavky jednotlivých procesů; další výhodou je velmi snadná realizace ‘asynchronních‘ akcí, kdy ve chvílích nečinnosti systému souborů může server provádět nějaké vlastní akce - statistiku, ukládání vyrovnávacích pamětí na disk, defragmentaci disku a podobně.

Soubory a bezpečnost

- Z bezpečnostního hlediska je v systému souborů nejpodstatnější mechanismus **přístupových práv**. Tento mechanismus určuje každému uživateli, jaké akce smí nebo nesmí provádět nad jednotlivými soubory a zabrání pokusům o provedení těch zakázaných. Např. systémové soubory tak mohou být pro uživatele (a samozřejmě tedy také uživatelské programy) zcela nepřístupné; díky tornu prakticky nepřipadá v úvahu, aby některý z uživatelů omylem nebo se zlým úmyslem jakkoli narušil operační systém. Podobně jsou chráněny i soubory jednotlivých uživatelů navzájem.

Přístupová práva v UNIXu

- Každý soubor obsahuje **identifikaci majitele** a **tři skupiny tří atributů**. První skupina se vztahuje k majiteli souboru; druhá skupina atributů určuje práva jisté vyhrazené skupiny uživatelů a třetí atributy platí pro všechny ostatní.
- Majitel souboru má jedno význačné privilegium: může měnit přístupová práva i identifikaci majitele.
- Přístupová práva v každé kategorii uživatelů jsou určena trojicí ‘povolení‘: **číst, psát a provést jako program**. Význam všech tří atributů je asi zřejmý; uvědomme si ale význam samostatného atributu pro čtení souboru. Znemožníme-li někomu číst obsah souboru, znamená to, že soubor nemůže prohlížet, číst nebo kopírovat; může jej však využít jinak - obsahuje-li např. soubor spustitelný program a má-li uživatel povolení tento soubor spustit, může jej volně používat. Nemůže však studovat jeho obsah a nemůže se tedy např. na základě jeho implementačních detailů pokusit proniknout do systému.

Příkaz ls

$ ls –la kde odezvou v našem případě bude

```
drwxr-xr-x   2   petr   group    48 May 12 10:37   .
drwxr-xr-x  16   bin    bin     256 May 12 10:37   ..
-rw-r------  1   petr   group  506 May 12 10:37   .profile
```

Každý řádek uvedeného výpisu se vztahuje vždy k jednomu souboru. První znak výpisu na řádku je typ souboru a může být

- \- obyčejný soubor,
- d adresář,
- l nepřímý odkaz,
- c znakový speciální soubor,
- b blokový speciální soubor.

Ve výpisu atributů souboru následuje dále devět znaků, které určují přístupová práva k souboru, přitom jednotlivé trojice postupně pro vlastníka souboru, skupinu, do které vlastník patří, a pro ostatní uživatele. V každé trojici mohou přístupová práva znamenat

- r čtení souboru je dovoleno,
- w zápis do souboru je dovolen,
- x obsah souboru může být spuštěn jako proveditelný program,
- \- přístup k souboru je zakázán.

Propůjčení identity

- Někdy je zapotřebí, aby mohl i běžný uživatel používat programy, které pracují se systémovými soubory - Často třeba chceme, aby mohl uživatel sám určovat heslo, pomocí kterého autorizuje svůj přístup k systému. hesla však pravděpodobně budou uložena v souboru přístupném pouze operačnímu systému - jak to tedy vyřešit?
- UNIX tento problém řeší tak, že zavádí možnost tzv. **propůjčeni identity**. Uživatel s dostatečnou prioritou může vytvořit potřebný program a uložit jej do souboru který vlastní; tento soubor pak označí speciálním atributem (s-bit místo x-bit) a zpřístupní jej pouze pro spuštění i ostatním uživatelům. Operační systém při spouštění takového programu interpretuje speciální atribut a programu po dobu běhu přidělí práva vlastníka souboru s programem, a ne uživatele, který jej skutečně spustil. Tento mechanismus je účinný, a při zachování určité ostražitosti ze strany systémových programátorům je i bezpečný. Princip je v tom, že je uživatel v akci regulován programem, který vlastní superuživatel. Nastavením s-bitu může také obyčejný uživatel dát k dispozici kontrolované manipulace se svými daty ostatním uživatelům.

Přístupová práva Novell Netware k souborům

Right	Syntax	Permissions Granted
Supervisory	S	All rights to the file. Modify the Inherited Rights Mask. Grant other users the Supervisory right.
Read	R	Open and read the file.
Create	C	Salvage the file after it has been deleted.
Write	W	Open and write to the file.
Erase	E	Delete the file.
Modify	M	Change the file's attributes and rename the file.
File Scan	F	See the filename when viewing the directory. See the directory structure, from the file to the root of the directory.
Access Control	A	Modify the file's trustee assignments and Inherited Rights Mask. Grant all file rights, except Supervisory, to other users.

Přístupová práva k adresářům

Right	Syntax	Permissions Granted
Supervisory	S	All rights to the directory, its files, and its subdirectories. Grant other users the Supervisory right. The Supervisory right overrides the Inherited Rights Masks and can be revoked only from the directory where it was granted.
Read	R	Open files in a directory and read their contents or run the programs.
Write	W	Open and modify files in the directory.
Create	C	Create files and subdirectories in the directory. Granting only Create at the directory level and no rights below the directory creates a "drop box" directory, where users can copy files but have no other rights in the directory.
Erase	E	Delete a directory, its files, its subdirectories, and its subdirectory files.
Modify	M	Change directory and file attributes. Rename the directory, its files, and its subdirectories.
File Scan	F	See directory files in a directory listing.
Access Control	A	Modify a directory's or a file's trustee assignments and Inherited Rights Mask. Grant any right (except Supervisory) to other users.

Atributy souborů

Option	Use to
ALL	Specify the A, Ci, Di, H, Ic, P, Ri, Ro, Sh, Sy and T attributes as a group. Primarily used to assign files these specific attributes.
A (Archive needed)	Indicate that the file has been modified since the last backup.
Ci (Copy Inhibit)	Prevent files from being copied. (Only for MAC files.)
Dc (Do not Compress)	Prevent a file from being compressed.
Di (Delete Inhibit)	Prevent a file from being deleted or copied over.
Dm (Don't Migrate)	Prevent a file from being migrated to a secondary backup system (regardless of what the volume or directory is set to).
Ds (Don't Suballocate)	Prevent an individual file from being suballocated, even if suballocation is enabled for the system. Use on files which are enlarged or appended to frequently, such as certain database files.
H (Hidden)	Prevent a filename from being displayed with the DOS DIR command. The file can't be copied or deleted.

Ic (Immediate compress)	Compress a file as soon as the OS can.
N (Normal)	Specify the Rw attribute.
P (Purge)	Purge a file immediately if the file is deleted.
Ri (Rename Inhibit)	Prevent a file from being renamed.
Ro (Read Only)	Allow a file to only be read; it can't be written to or deleted (in other words, Ro includes Ri and Di).
Rw (Read Write)	Allow a file to be read and written to.
Sh (Shareable)	Allow a file to be used by several users simultaneously.
Sy (System)	Prevent a filename from being displayed with the DOS DIR command. It can't be copied or deleted.
T (Transactional)	Protect a file by using the Transaction Tracking System™.
X (Execute Only)	Prevent a file from being copied or copied over. This attribute can be given only to .EXE or .COM files, *and cannot be removed.*

Atributy adresářů

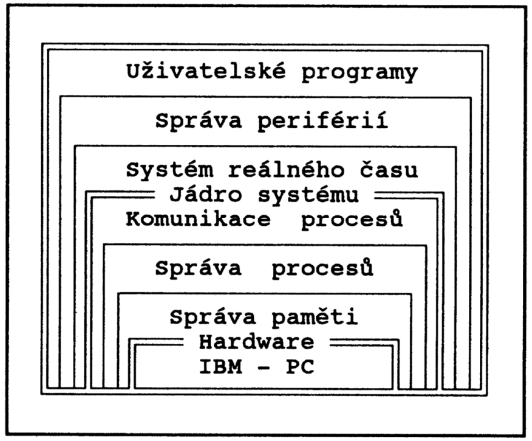
Option	Use to
ALL	Specify the Di, H, Ic, P, Ri, and Sy attributes as a group. Primarily used to assign directories these specific attributes.
Di (Delete Inhibit)	Prevent the directory from being deleted.
Dm (Don't Migrate)	Prevent the directory from being migrated to a secondary backup system (regardless of what the volume is set to).
H (Hidden)	Prevent the directory from being seen with a DOS DIR command.
Ic (Immediate Compress)	Compress the files in the specified directory as soon as the OS can. This does not apply to the directory's subdirectories and the files in them. Use Ic in combination with the /S option to apply immediate compression to the directory's subdirectories and their files recursively.
N (Normal)	Specify no attributes.
P (Purge)	Purge the directory or files in a directory immediately when deleted.
Ri (Rename Inhibit)	Prevent the directory from being renamed.
Sy (System)	Prevent the directory from being seen with a DOS DIR command; also to prevent it from being copied or deleted.

Počítačové sítě

- V dávné minulosti operační systémy nepodporovaly vzájemnou komunikaci mezi počítači. Počítače byly izolované výpočetní jednotky.
- První náznak aparátu pro komunikaci byl v systému UNIX tzv. program **uucp** (unit to unix communication program) přes sériovou linku.
- Po vzniku počítačových sítí se komunikace přes tuto síť začala objevovat i jako nedílná součást operačního systému (existovala celá řada technologií a architektur)
- S současné době převážila architektura **TCP/IP**. Operační systém bez podpory této architektury by byl prakticky neprodejný a nepoužitelný.

- Součástí počítačové sítě je samozřejmě především technické vybavení, které umožňuje vlastní propojení počítačů. Nejjednodušší síť může pracovat s obyčejným sériovým rozhraním; výhodou takové sítě je relativní jednoduchost, nevýhodou velmi nízká přenosová rychlost. Naprostá většina dnešních lokálních sítí disponuje speciálním technickým vybavením, které průchodnost sítě zvyšuje o několik řádů.
- Snad ještě důležitější než kvalita technického vybavení je pro síť kvalita programového vybavení, které uživatelům sítě nabízí potřebné služby. Vyšší vrstvy tohoto vybavení určují kvalitu a různorodost služeb sítě (snad všechny sítě zajišťují sdílení souborů, časté je sdílení dalších prostředků - tiskáren, modemů nebo faxů; kvalitnější síťové vybavení však může sloužit i pro komunikaci mezí programy nebo třeba pro distribuované zpracování úloh).
- Ještě daleko významnější je však úloha nižších vrstev síťového programového vybavení - právě ty totiž určují spolehlivost a průchodnost sítě.
- Návrh lokální sítě je totiž problematický proto, že se v něm musíme vypořádat se skutečným paralelismem na mnoha úrovních: práce procesů na jednom počítači je plně paralelní vůči práci zařízení, které zajišťuje síťový přenos; nezávisle na obou pracuje zařízení zabezpečující síťový přenos na jiném počítači v síti i jeho procesy.
- Všechny tyto prvky musíme synchronizovat tak, aby jejich spolupráce probíhala bez chyb, a přitom aby systémy nestrávily více času vzájemnou komunikací než zpracováváním programů.

Vrstvená struktura OS

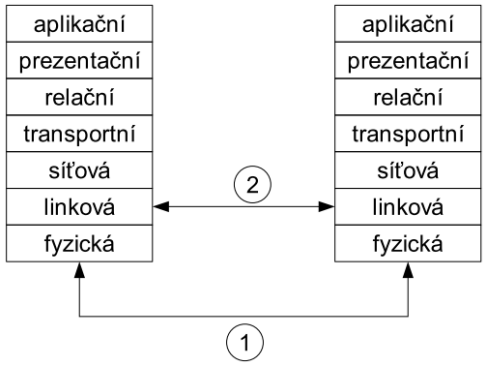


Kam vložit vrstvu síťového programového vybavení?

ISO/OSI referenční model

- ISO – International Standards Organization
- OSI – Open System Interconnection
- mezinárodní standard pro počítačové sítě
- popisuje způsoby jak lze připojit různé zařízení za účelem vzájemné komunikace
- má sedmivrstvou architekturu, definovanou v normalizačních materiálech ISO
- 7 vrstev tvoří hierarchii (posloupnost) začínající fyzickým spojením na nejnižší úrovni a končící aplikacemi na nejvyšší úrovni
- každá vrstva přijímá data od nižší vrstvy, upravuje je a předává data vrstvě vyšší (resp. opačně)
- ke každé vrstvě přísluší rozhraní se sousedící vrstvou
- odděluje síťový hardware a software

Komunikace uzlů

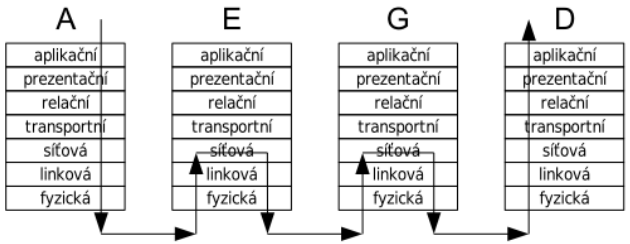


1 Fyzické propojení uzlů
2 Komunikace stejnolehlých vrstev dle tzv. **protokolu**

- **Fyzická vrstva (Physical Layer)**
  - zajišťuje přenos jednotlivých bitů mezi příjemcem a odesílatelem
  - zabývá se technickými, hardwarovými vlastnosti, délku jednotlivých bitů, úrovně napětí
  - pasivní a aktivní prvky
  - síťové karty; přenosové média
- **Linková vrsta (Data Link Layer)**
  - zajišťuje bezchybný přenos celých bloků dat (rámců - frames)
  - musí rozeznat: začátek, konec rámce a jednotlivé části
- **Síťová vrstva (Network Layer)**
  - zajišťuje směrování přenášených paketů
  - volbu trasy přes mezilehlé uzly a postupné předávání jednotlivých paketů od odesílatele k příjemci
  - musí znát topologii sítě

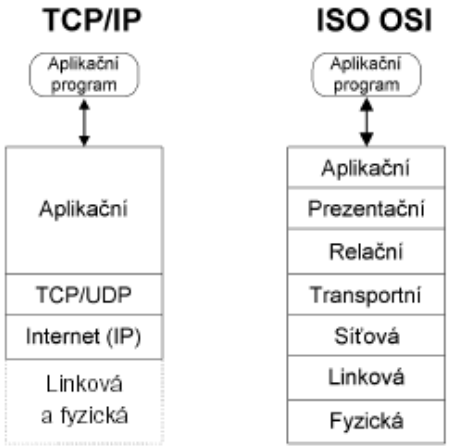
- **Transportní vrstva (Transport Layer**)
  - vytváří přímé spojení mezi koncovými účastníky
  - při odesílání dat sestavuje jednotlivé pakety do kterých rozděluje přenášené data
  - na straně příjemce se skládá do původního tvaru
- **Relační vrstva (Session Layer)**
  - navazuje, udržuje a ruší relace mezi koncovými účastníky
- **Prezentační vrstva (Presentation Layer)**
  - zabývá se konverzí, kompresí a kódováním
- **Aplikační vrstva (Application Layer)**
  - předává jednotlivých aplikacím (spuštěným) data

Přechod dat přes routery



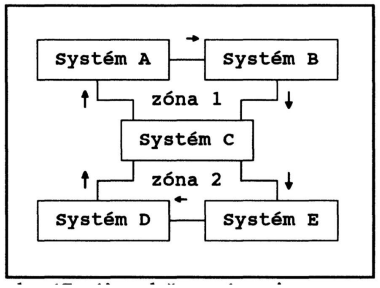
Prvky E a G jsou tzv. směrovače (routers). Nemusí být implementovány všechny vrstvy

ISO/OSI versus TCP/IP

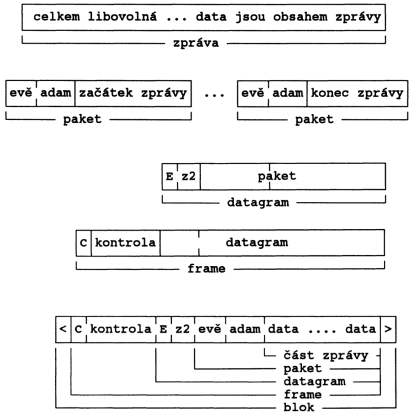


Topologie sítě se 2 zónami

- Systém C je zapojen v obou zónách a zajišťuje předávání dat mezi nimi – tzv. router
- Směr toku dat v síti ukazují šipky
- Princip průchodu dat jednotlivými vrstvami – Vrstva si k datům, které převzala od vyšší vrstvy, přidá tzv. hlavičku (doplňující informace nutné pro správnou činnost vrstvy) a takto vytvořený blok dat předá nižší vrstvě
- Při průchodu vrstvami opačným směrem se odebírají hlavičky a data se předají vyšší vrstvě

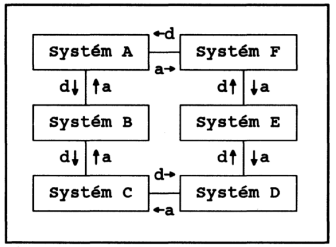


Průchod dat vrstvami



Počítačová síť XINU

- Počítač s operačním systémem XINU, který se má stát součástí sítě, musí být vybaven dvěma sériovými linkami.
- Prostřednictvím každé z nich je propojen s nejbližším dalším uzlem; ‘poslední’ uzel je propojen opět s prvním - topologie celé sítě je tedy kruhová.
- Všechny sériové linky jsou obousměrné; nejsou však používány oběma směry pro tok dat. Data jsou v síti předávána pouze jedním směrem (který naznačují šipky s označením **d** v obrázku), druhým směrem chodí potvrzení o korektním přijetí jednotlivých paketů (tomu odpovídají šipky s označením **a**).
- Tato potvrzení, generovaná síťovou vrstvou, jsou velmi důležitá pro spolehlivost sítě.



Problémy implementace sítě

- Pro alokaci paměti potřebné pro odesílaná a přijatá data budeme používat služby `new` a `dispose`. Služba `new` vždy paměť alokuje; není-li paměti dostatek, je volající proces pozastaven, dokud paměť nebude k dispozici.
- Je však nutné implementovat obě služby tak, aby alokovaly paměť ze samostatného bloku nepřístupného běžným službám alokace paměti. Síťový přenos je totiž často rychlejší než zpracování přijatých dat; pokud by příjemce nebyl omezen, mohl by alokovat paměť pro další a další přijaté bloky tak dlouho, až by vyčerpal všechnu volnou paměť v systému a došlo by k zablokování systému z důvodu vyčerpání volné paměti.

- Buffer, který se používá pro předávání dat, je vždy bufferem pro frame. Vyšší vrstvy tedy v tomto bufferu vyplňují pouze část, hlavičky nechávají volné pro pozdější doplnění nižšími vrstvami.
- Důvod je jednoduchý - kdyby měla každá vrstva kopírovat data do svého vlastního bufferu, byl by ovladač sítě příliš neefektivní. Takto prochází jediný buffer všemi vrstvami; ty si předávají pouze ukazatel na jeho začátek.
- **Linková vrstva** sítě není ničím jiným, než **ovladačem zařízení**; rozhraní mezi ní a síťovou vrstvou je tedy standardním rozhraním ovladačů zařízení. Rozhraní mezi ostatními vrstvami však musíme teprve definovat.

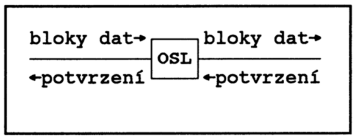
- Velmi dobře se pro komunikaci mezi jednotlivými vrstvami síťového software hodí aparát zpráv; síťová vrstva bude využívat zpráv pro komunikaci s relační vrstvou.
- Systém zpráv, popsaný dříve, je však příliš jednoduchý a jeho nedostatky (např. nemožnost odeslat více zpráv jedinému procesu, který by si je pak mohl postupně odebírat) jej pro využití v tomto případě diskvalifikují.
- V novém systému zpráv nebude příjemcem zprávy proces, ale nový objekt – tzv. **port**. Port je identifikován svým číslem; každý proces, který zná číslo portu, na něj může odeslat zprávu (která obsahuje adresu bufferu). Podobně každý proces, který zná číslo portu z něj může zprávu přečíst - je-li nějaká k dispozici. Pokud žádná zpráva k dispozici není, je čtoucí proces pozastaven až do chvíle, kdy někdo zprávu na port zapíše. Pokud po nějakou dobu nikdo zprávy z portu nečte, dokáže port spravovat frontu několika zpráv; je-li fronta již plná a některý proces se pokusí odeslat na port další zprávu, je pozastaven až do doby, kdy někdo nějaké zprávy z portu odebere.
- Pro práci s porty se používají služby:

```
void psend( int port, void *msg );
void *preceive( int port);
```

Linková vrstva

- Software pro linkovou vrstvu síťového vybavení není ničím jiným, než ovladačem sériového rozhraní. Oproti obecnému ovladači je zde však řada rozdílů:
  - Ovladač síťové linky nemusí být schopen obsloužit více procesů, protože k němu přistupují pouze rutiny vyšších vrstev síťového vybavení. Ovladač proto nevyužívá semafory pro synchronizaci procesů - jednoduše předpokládá, že zatímco obsluhuje požadavek jednoho procesu (který je na tu dobu pozastaven), jiný proces jeho služby volat nebude.
  - Zatímco obecný ovladač předával přijatá data po jednotlivých bytech, bude ovladač sítové linky předávat po lince celé bloky. Z toho důvodu také nepotřebuje zvláštní buffer na ukládání přijatých nebo ještě neodeslaných bytů; namísto toho využívá přímo bufferu, ve kterém je uložen frame.

- Zdálo by se tedy, že ovladač síťové linky bude daleko jednodušší než obecný ovladač. To však je pravda jen pro vlastní přenos; konstrukci celého ovladače komplikuje několik dalších faktorů:
  - Síťový ovladač má dvě linky, jednu pro vstup a druhou pro výstup dat. Opačný směr obou linek využívá síťová vrstva pro předávání potvrzení o tom, že frame přišel a je v pořádku nebo naopak - frame nepřišel vůbec, nebo přišel a je poškozen.
  - Tuto situaci ilustruje obrázek. OSL je ovladač síťové linky; obě linky vedou jedním směrem datové bloky a druhým potvrzení. Ačkoli potvrzení by mohla být odesílána ve formě speciálních datových bloků (a u složitějších sítí tomu tak skutečně bývá), je daleko jednodušší využít pro potvrzení jediný byte.
  - Z toho však vyplývá, že ovladač síťové linky musí být schopen pracovat ve dvou režimech - v blokovém a v bytovém. Po jedné lince na obrázku ji vidíme vlevo - ovladač přijímá data v blokovém režimu a odesílá potvrzení jako byty. U druhé linky je tomu právě opačně - data jsou odesílána po blocích, ale příjem dat pracuje v bytovém režimu.



  - Pro blokový režim použijeme standardní služby systému ovladačů `read` a `write`. Pro odesílání bytů v bytovém režimu můžeme pohodlně použít službu `putc` a pro jejich příjem službu `getc`.
  - Zatímco však služba `putc` může být naprogramována poměrně jednoduše - prostě odešleme znak - se službou `getc` je situace složitější: nemáme k dispozici ani buffer, ani vstupní semafor, musíme si proto pomoci jinak.
  - Ideální je pro tento účel mechanismus zpráv. Služba `getc` bude čekat na příjem zprávy; tu jí pošle dolní polovina ovladače po přijetí znaku, obsahem zprávy bude právě přijatý znak.
- Další problém vychází z toho, že přenos síťového bloku probíhá poměrně velmi rychle; pokud by někde uprostřed přenosu došlo k přeplánování, ztratilo by se několik bytů a přenos by byl chybný. Musíme proto využít mechanismus pozastavení hodin. Ten zajistí, že časovač po dobu přenosu bloku nevyvolá přeplánování.
- Pozastavení hodin však pro dokonalé vyřešení problému nestačí. Příčinou přeplánování totiž nemusí být pouze časovač; může k němu dojít také z řady jiných důvodů z iniciativy aktivního procesu. Abychom přeplánování po dobu přenosu s jistotou zabránili, musíme kromě pozastavení hodin také dočasně přidělit nulovému procesu velmi vysokou prioritu, takže po celou dobu poběží právě on.
- Pro zajištění maximální průchodnosti sítě a zároveň maximální výkon systému musíme mít k dispozici speciální síťové technické vybavení, které přenáší bloky nezávisle na činnosti procesoru a disponuje vyrovnávací pamětí, takže přeplánování nebude vadit.

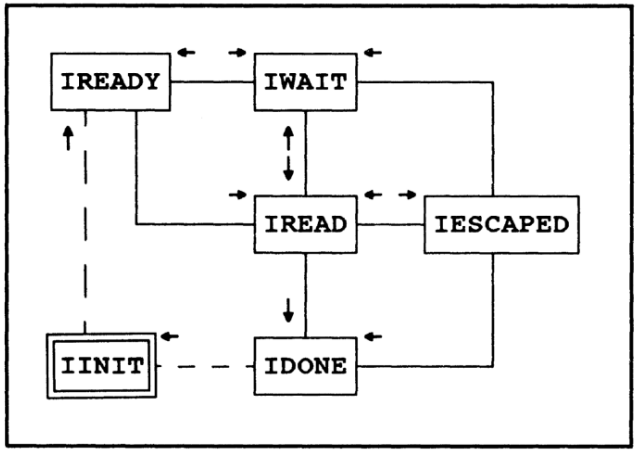
- To, že příjemce na začátku příjmu bloku přepne na nulový proces, přece jen nějakou chvilku trvá - je nutné nulovému procesu zvýšit prioritu a přeplánovat, ve většině případů součástí přeplánování i přepnutí kontextu. Pokud bychom pro přenos bloků používali opravdu rychlé sériové rozhraní, mohlo by mezitím příjemci utéci několik prvních bytů z bloku. Odesílající proto musí po odeslání speciálního znaku pro začátek bloku chvíli počkat, a pak teprve může začít odesílat frame.
- Odesílající však je implementován jako obslužná rutina přerušení; v takovém případě není ‘chvíli počkat‘ nic jednoduchého:
  - Nemůžeme využít standardní mechanismus služby `sleep`, protože obslužná rutina přerušení může běžet v rámci nulového procesu a ten službu `sleep` využívat nesmí.
  - Nelze použít ani čekací smyčku - při běhu obslužné rutiny přerušení jsou další přerušení zakázána; došlo by tak k absurdní situaci, kdy by čekání odesílací rutiny síťového ovladače (mimo jiné) znemožnilo funkci jeho přijímací části, která je pochopitelně na přerušeních závislá.
  - Pro vyřešení tohoto problému musíme využít technické vybavení. Dolní polovina ovladače odešle speciální znak pro začátek bloku a pak ukončí práci; nejprve však inicializuje technické zařízení, které zajistí, že po určitém čase bude aktivováno přerušení, které obslužnou rutinu znovu aktivuje. Potom teprve obslužná rutina odešle vlastní blok.

- Posledním zásadním problémem je rozumné ošetření situace, kdy přijímací rutina sama detekuje nějakou chybu při přenosu (chyba parity PE, chyba rámce FE, chyba přeběhu OE – tyto chyby hlásí hardware). Nejjednodušší by samozřejmě bylo takovou chybu prostě ohlásit síťové vrstvě, která by ji obsloužila stejným způsobem, jako chybu zjištěnou při kontrole obsahu frame. Bylo by to jednoduché; zároveň však velmi neefektivní - zvláště v případě, že by se chyba objevila na samém začátku přenášeného bloku.
- Přijímací rutina se proto pokusí vyžádat si od vysílajícího okamžité nové odeslání bloku. Jestliže tento požadavek vysílací rutina stačí zachytit dříve, než odešle celý blok a skončí, odešle prostě blok znovu. Jinak zůstane odstranění chyby skutečně až na síťové vrstvě; to již však nebude znamenat takové zpomalení, protože chyba se jistě objevila až ke konci bloku (jinak by požadavek na nové odeslání přišel včas).

- Přijímající musí být schopen spolehlivě rozpoznat speciální kódy pro začátek a konec bloku; jsou-li tyto kódy odesílány jako běžné byty (a při použití sériového rozhraní nemáme jinou možnost), musíme zajistit, aby se stejné kódy nemohly vyskytnout uvnitř bloku. Řešení je celkem jednoduché: zavedeme navíc tzv. **escape** znak, který mění význam následujícího znaku. Místo bytu, který by měl stejnou hodnotu jako některý ze speciálních kódů, odešleme dvojici **escape** znak + změněný byte; přijímající část tuto dvojici opět přeloží zpátky na původní byte.
- Návrh ovladače síťové linky je velmi citlivý na jakékoli nedomyšlenosti a chyby. I nepatrný nedostatek může být snadno příčinou významného snížení průchodnosti sítě nebo dokonce jejího úplného zablokování. Může např. dojít ke klasickému zablokování, kdy několik systémů čeká vzájemně na data a/nebo potvrzeni, a ani jeden z nich se nedočká - to je síťová varianta známého deadlocku. Jinou nebezpečnou situací je stav, kdy se systémy snaží sesynchronizovat, ale nikdy se jim to nepodaří - mohlo by např. dojít ke ztrátě potvrzení bloku; vysílající pak bude blok neustále posílat znovu, zatímco příjemce jej nebude akceptovat, protože již čeká na blok následující.

- Pro návrh ovladače je nejvhodnější použít metodu **stavového diagramu**
- Jedná se vlastně opět o modularitu, tentokrát přenesenou až dovnitř do jediné funkce:
  - navrhovaná rutina se může nacházet v několika stavech, každý odpovídá jiné situaci; potřebné chování rutiny pak zkoumáme pro každý slav zvlášť. Tím si celý problém rozdělíme na několik problémů menších a jednodušších, a snáze se vyhneme případné chybě nebo tomu, že bychom na něco zapomněli.
  - Další výhodou stavového diagramu je to, že se velmi snadno přenáší do skutečného programu - stačí využít statickou proměnnou obsahující identifikaci momentálního stavu, a přechody mezi jednotlivými stavy pak realizovat změnou obsahu této proměnné.

Stavy vstupního ovladače



Vstupní ovladač

- Stav *IINIT* je základním stavem, ve kterém se ovladač nachází, když se nic neděje. Mohou v něm být přijímány jednotlivé znaky v neblokovém režimu.
- Do stavu *IREADY*. ve kterém je ovladač připraven ke čtení bloku, se ze stavu *IINIT* dostane při volání vstupní služby *read* vyšší vrstvou síťového programového vybavení.
- Po přijetí speciálního kódu pro začátek bloku se ovladač dostane ze stavu *IREADY* do stavu *IREAD*, ve kterém postupně čte jednotlivé byty bloku a ukládá je do paměti. Přijme-li ovladač znak ‘escape‘, neukládá nic, ale přejde do stavu *IESCAPED*; ten se od stavu *IREAD* liší pouze tím, že přijatý byte navíc překóduje, uloží a vrátí se do stavu *IREAD*.
- Příchod speciálního znaku pro ukončení bloku převede ovladač do stavu *IDONE*. Ten nemá žádný zvláštní význam, ovladač v něm pouze čeká, než jej ukončení služby *read* opět převede do stavu *IINIT*.

Stav IINIT

- Ve stavu *IINIT* může být ovladač aktivován jedině v případě, že přišel nějaký znak v neblokovém režimu, tj. odeslaný službou `putc` (tento ovladač tedy datové bloky pouze vysílá, přijímá jen jednotlivé znaky). Ovladač musí rozlišit dvě možnosti:
  - jedná-li se o požadavek linkové vrstvy druhého počítače na nové odeslání bloku (ten je identifikován speciální hodnotou znaku), předá se tento požadavek výstupnímu ovladači.
  - jinak se jedná o vzájemnou 'potvrzovací' komunikaci síťových vrstev; znak proto předá službě `getc` pomocí mechanismu zpráv.
- V obou případech ovladač zůstane ve stavu *IINIT*.

Stav IREADY

- Ve stavu *IREADY* ovladač čeká na začátek přenosu bloku. Může dojít k těmto situacím:
  - Objeví se speciální znak pro začátek bloku. Ovladač pozastaví hodiny, zvýší prioritu nulového procesu, zajistí případné přeplánování, připraví se na příjem bloku a přejde do stavu *IREAD*.
  - Objeví se jakýkoli jiný znak. To zřejmě znamená, že vysílající 'vypadl ze synchronizace' a právě odesílá nějaký blok. Ovladač odešle požadavek na nové odeslání bloku a přejde do stavu *IWAIT*.

Stav IREAD

- Stav IREAD je komplikovanější - zde musíme rozlišit více různých případů:
  - Jestliže došlo k chybě při přenosu (to ohlásí technické vybavení), přeruší rutina příjem bloku, vrátí nulovému procesu jeho nízkou prioritu, spustí opět pozastavené hodiny, odešle po lince požadavek na opakování bloku a přejde do stavu *IWAIT*.
  - Po přijetí speciálního znaku pro konec bloku rutina aktivuje proces, který (uvnitř služby `read`) na blok čekal, vrátí nulovému procesu jeho nízkou prioritu, spustí opět pozastavené hodiny a přejde do stavu *IDONE*.
  - Po přijetí speciálního znaku pro začátek bloku prostě rutina zahodí dosud načtené znaky a začne znovu přijímat blok od začátku. Stav se v tomto případě nemění (je možné jej samozřejmě nastavit opět na *IREAD* pro větší shodu se stavem *IESCAPED*).
  - Po přijetí speciálního znaku 'escape' rutina přejde do stavu *IESCAPED.*
  - Přišel-li jakýkoli jiný znak, uloží jej rutina do přijímaného bloku. Aby nedošlo k přetečení paměti vyhrazené pro blok, musí rutina zkontrolovat, není-li již buffer zaplněn; pokud tomu tak je, ukončí se přenos stejně jako po přijetí znaku pro konec bloku

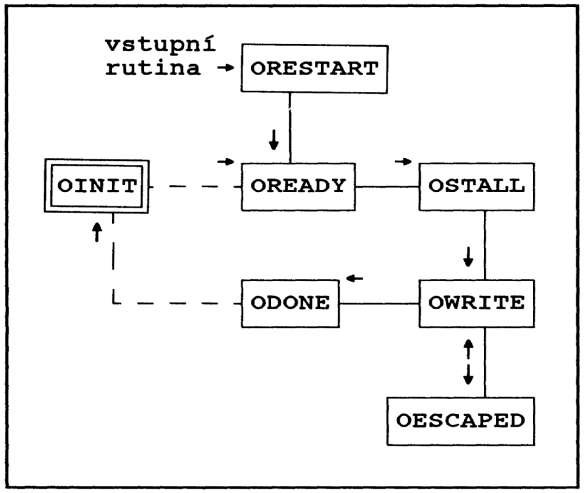
Stav IESCAPED

- Stav *IESCAPED* se do značné míry podobá stavu *IREAD* - v případě chyby při přenosu, přijetí speciálního znaku pro konec bloku nebo speciálního znaku pro začátek bloku se chová stejně (po zachycení začátku bloku navíc pouze změní stav na *IREAD*). Jediný rozdíl spočívá v tom, že:
  - Přijetí znaku 'escape' je považováno za chybu při přenosu a rutina se tedy chová stejně, jako když dojde ke skutečné chybě;
  - Jakýkoli jiný znak se před uložením překóduje.
- V praxi je výhodné implementovat stavy *IESCAPED* a *IREAD* společně;

Stavy IWAIT A IDONE

- Ve stavu *IWAIT* rutina rozlišuje tři možné případy:
  - Objeví-li se speciální kód pro konec bloku, přejde rutina do stavu *IREADY*.
  - Po přijetí speciálního kódu pro začátek bloku se rutina chová stejně, jako by byla ve stavu *IREADY*: pozastaví hodiny, zvýší prioritu nulového procesu, zajistí případné přeplánování, připraví se na příjem bloku a přejde do stavu *IREAD*.
  - Jakýkoli jiný znak se ignoruje beze změny stavu.
- Nejjednodušší je chování rutiny v posledním stavu *IDONE*. V něm rutina jakýkoli znak ignoruje a jen čeká, až bude horní polovinou ovladače převedena do stavu *IINIT*.

Výstupní ovladač



Výstupní ovladač

- Stav *OINIT* je základním stavem, ve kterém se ovladač nachází když ‘se nic neděje‘. V tomto stavu rutina zajistí vysílání jednotlivých znaků v neblokovém režimu, je-li to zapotřebí
- Do stavu *OREADY* se ze stavu *OINIT* ovladač dostane při volání výstupní služby `write` vyšší vrstvou síťového programového vybavení. Ovladač zajistí odeslání speciálního znaku pro začátek bloku a přejde do stavu *OWRITE*.

- Ve stavu *OWRITE* ovladač odešle byte po bytu celý blok. Je-li zapotřebí odeslat byte, který odpovídá některému ze speciálních znaků, ovladač odešle nejprve znak ‘escape‘ a pak překódovaný byte. Formálně tuto situaci vyřešíme pomocným stavem *OESCAPED*; Po odeslání celého bloku ovladač odešle speciální znak pro ukončení bloku a přejde do stavu *ODONE*.
- Ve stavu *ODONE* již ovladač pouze čeká, než jej ukončení služby `write` opět převede do stavu *OINIT*
- Z technických důvodů musíme k tomuto ‘ideálnímu návrhu‘ přidat ještě dva další stavy. Stav *OSTALL* je vsunut mezi stavy *OREADY* a *OWRITE* a ve spolupráci s technickým vybavením zajišťuje pauzu mezi odesláním hlavičky bloku a vlastních dat bloku. Do stavu *ORESTART* je ovladač přemístěn z iniciativy vstupní rutiny po přijetí požadavku na nové odeslání bloku; ovladač v tomto stavu ‘zapomene‘ co již odeslal, a přejde do stavu *OREADY*.

Stavy OINIT, OREADY, OSTALL

- Ve stavu *OINIT* může být ovladač aktivován ve dvou případech:
  - Je zapotřebí odeslat znak v neblokovém režimu (tento ovladač tedy datové bloky přijímá). Ovladač znak odešle.
  - Technické vybavení hlásí ukončení přenosu a již není co odesílat. Ovladač deaktivuje zařízeni, takže to již další přerušení nebude generovat (dokud jej někdo opět neaktivuje).
  - V obou případech ovladač zůstane ve stavu *OINIT*.

- Ve stavu *OREADY* ovladač prostě odešle speciální znak pro začátek bloku a přejde do stavu *OSTALL*.

- Stav *OSTALL* spolupracuje s technickým vybavením; jeho obsah tedy silně závisí na konkrétním počítači. V originálním XINU, psaném pro mikropočítač LSl 11, se ve stavu 'OSTALL' zařízení pro linkový přenos nejprve deaktivuje a hned nato opět aktivuje. To u zmíněného systému zajistí vygenerování dalšího přerušení technickým vybavením. Ovladač ve stavu *OSTALL* jen počítá tato 'zbytečná' přerušení; je-li jich dostatek (trvá-ti tedy pauza již dost dlouho), přejde do stavu *OWR ITE*

Stavy OWRITE, ODONE, ORESTART

- Ve stavu OWRITE rutina prostě odešle další znak bloku; je-ti to zapotřebí, zakóduje jej ve spolupráci se stavem *OESCAPED*.
- V praxi je opět výhodné implementovat stavy *OESCAPED* a *OWRITE* společně;
- Jedině ve stavu *ODONE* je výstupní rutina nepatrně složitější než vstupní. Musí v něm totiž nejprve aktivovat proces, který čekal na odeslání bloku, a pak deaktivovat linkové zařízení. Pak již rutina opět jen čeká, až bude horní polovinou ovladače převedena do stavu *OINIT*.
- Poslední stav *ORESTART* pouze znovu inicializuje odeslání bloku a ihned přejde do stavu *OREADY*. Protože stav *ORESTART* sám nic neodesílá, může do stavu *OREADY* přejít skutečně ihned, tedy ne až po přijetí dalšího přerušení, jako tomu je v ostatních případech stavových přechodů.

Společné deklarace

```
/* netlink.h */

enum {                  /* stavy vstupní rutiny */
IINIT, IREADY, IREAD, IESCAPED, IWAIT, IDONE
};

enum {                  /* stavy výstupní rutiny */
OINIT, OREADY, OSTALL, OWRITE, OESCAPED, ODONE, ORESTART
};

#define NLESC           ' \xa8'     /* escape znak */
#define NLRESTART       ' \xa9'   /* požadavek na nové odeslání */
#define NLSTART         '\xaa'      /* začátek bloku */
#define NLEOB           '\xab'    /* konec bloku */
#define NLSPEC          '\xa8'     /* bity určující spec. znaky */
#define NLSMASK         '\xfc'    /* maska pro —— " —— ,
#define NL_DOESC        '\x7f '   /* maska pro zakódování po escape * /
#define NL_UNESC        '\x80 '  /* ' odkódování ' po escape */
#define NSTALL       3        /* počet průchod stavem OSTALL */
```

```
struct nlblk {
        char istate;            /* stav vstupního ovladače */
        int icount;             /* počet načtených znaků */
         int ipid;              /* id procesu čekajícího na vstup */
        char *inext;            /* první volná pozice v bufferu */
        char *istart;           /* buffer */
        int imax;               /* max. délka čteného bloku */
        char ostate;            /* stav výstupního ovladače */
        int ocount;             /* počet odeslaných znaků */
        int opid;               /* proces čekající na výstup */
        char *onext;            /* první neodeslaný znak v bufferu */
        char *ostart;           /* buffer */
        int olen;               /* celkový počet znaků */
        int ostall;             /* čítač čekání ve stavu OSTALL */
        int cpid;               /* proces čekající na znak. vstup */
        char cchar;             /* znak pro znakový výstup */
        char cvalid;            /*!=0  je-li cchar validní */

        void (*run_odev)(void);         /* aktivace výstupního zařízení */
        void (*stop_odev)(void) ;       /* deaktivace výstupního zařízení */
        void (*wr_odev)(char);          /* zápis znaku na výst. zařízení */
        char (*rd_idev)(void);          /* čtení znaku ze vstup. zařízení */
        int (*err_idev) (void);         /* zjištění chyby vstup. zařízení */
};
extern struct nlblk nldevs[];
```

```
/* nlread.c */

#include <net_link.h>
#include <conf.h>
/*-----------------------------------------------
 * nl_read požadavek na čtení bloku
 *-----------------------------------------------
*/
int nl_read(struct devsw *devptr, char *buff, unsigned max)
{
        struct nlblk  *nl=nldevs [ devptr->dvminor];
        int nread;
        char x;

        sdisable(x);
        nl->istate=IREADY;
        nl->inext=nl->istart=buff;
        nl->imax=max;
        nl->icount=0;
        nl->ipid=currpid;
        suspend(currpid);
        nread=nl->icount;
        nl->istate=IINIT;
        restore(x);
        return(nread);
}
```

```
/* nl_write.c */
#include <net_link.h>
#include <conf.h>
/*-------------------------------------------------------------
 * nl_write -- požadavek na odeslání bloku po síti
 *
*/-------------------------------------------------------------
int nl_write(struct devsw *devptr, char *buff, unsigned len)
{
        struct nlblk *nl=nldevs [ devptr->dvminor];
        char x;

        sdisable(x);
        nl->ostate=OREADY;
        nl->onext=nl->ostart=buff;
        nl->olen=nl->ocount=len;
        nl->opid=currpid;
        nl->run_odev();
        suspend(currpid);
        nl->ostate=OINIT;
        restore(x);
        return(0);
}
```

```
    /* nl_putc.c */
    #include <net_link.h>
    #include <conf.h>
    /*
     *-----------------------------------------------
     * nl_putc  -- požadavek na odeslání bytu
     *-------------------------------------------------------
    */

    int nl_putc(struct devsw *devptr, char c)
    {
            struct nlblk *nl=nldevs [ devptr->dvminor];
            char x;

            sdisable(x);
            nl->cchar=c;
            nl->cvalid=1;
            nl->run_odev();
            restore(x);
            return(0);
    }
```

```
        /* nl_getc.c */
        #include <net_link.h>
        #include <conf.h>
        /*
          * nl_getc -- čtení jednoho bytu ze sítě
        */

        int nl_getc(struct devsw *devptr)
        {

                struct nlblk *nl=nldevs[devptr->dvminor];
                char x;

                sdisable(x);
                recvclr ();
                nl->cpid=currpid;
                restore(x);
                return(0);
        }
```

Použití služby nl_getc

- Služba nejprve smaže případnou zprávu službou `recvclr`, pak vyplní potřebné tabulky, aby byla zpráva odeslána patřičnému procesu, a pak - skončí. Proces si na zprávu musí počkat sám voláním služby `receive`.
- Tento netradiční mechanismus byl zvolen proto, že proces musí nejprve inicializovat čtení potvrzení, tedy provést službu `getc`, potom odeslat blok a až nakonec čekat na potvrzení. Pořadí služeb tedy bude

```
nl_getc();        /* zatím nic nečte! */
nl_write();
receive ();
```

- Pokud bychom se pokusili přidat službu `receive` na konec funkce `nl_getc` a použít pak (na první pohled logičtější) pořadí

```
nl_write();
nl_getc();
```

bylo by to špatně - odesílání bloků a potvrzení probíhá do jisté míry asynchronně, takže zprávu bychom mohli ve skutečnosti dostat mezi ukončením služby nl_write a začátkem služby `nl_getc`. Ta by pak sama zprávu zahodila službou `recvclr`.

```
/* nl_ohandler.c */
#include <net_link.h>
#include <conf.h>

void interrupt nl_ohandler()
{
        struct nlblk *nl=nldevs[devtab[NLDEV] .dvminor];

        switch (nl->ostate) {
                case OINIT:
                        if (nl->cvalid) {
                                nl->cvalid=0;
                                nl->wrt_odev(nl->cchar);
                        } else nl->stop_odev();
                        return;
                case ORESTART:
                        nl->ostate=OREADY;
                        nl->onext=nl->ostart;
                        nl->ocount=nl->olen;
                        /* fall thru */
                case OREADY:
                        nl->wrt_odev(NLSTART);
                        nl->ostate=OSTALL;
                        nl->ostall=NSTALL;
                        return;
```

```
        case OSTALL:
                nl->stop_odev();
                nl->run_odev();
                if (nl->ostall-- == 0)
                        nl->ostate=OWRITE;
                return;
        case OWRITE:
        case OESCAPED:  {
                char ch;
                if (nl->ocount-- == 0) {
                        nl->wrt_odev(NLEOB);
                        nl->ostate=ODONE;
                        return;
                }
                if ( ( (ch=*nl->onext++) & NLSMASK) == NLSPEC)
                        if (nl->ostate == OWRITE)  {
                                nl->onext--;
                                nl->ocount++;
                                nl->ostate=OESCAPED;
                                nl->wrt_odev(NLESC);
                        } else {
                                nl->ostate=OWRITE;
                                nl->wrt_odev(ch & NL_DOESC);
                        }
                else nl->wrt_odev(ch);
                }
                return;
```

```
                case ODONE:
                        if (nl->ocount<0) {
                                ready ( nl->opid, RESCHYES);
                                nl->ocount=0;
                        }
                        nl->stop_odev();
        }
}
```

```
/* nl_ihandler.c */
#include <net_link.h>
#include <conf.h>

void interrupt nl_ihandler()
{
        struct nlblk *nl=nldevs[devtab[NLDEV] .dvminor];
        char ch=nl->rd_idev();

        switch (nl->istate) {
                case IINIT:
                        if (ch==NLRESTART &&  nl->ostate != OINIT)
                                nl->ostate=ORESTART;
                        else if (nl->cpid)
                                sendf(nl->cpid, ch);
                        return;
                case IREADY:
                        if (ch==NLSTART) {
                SetRead:
                                nl->istate=IREAD;
                                stopclk();
                                nullprio(0,32767);
                        } else {
                                nl->cchar=NLRESTART;
                                nl->cvalid=1;
                                nl->run_odev();
                                nl->istate=IWAIT;
                        }
                        return;
```

```
                case IREAD:
                case IESCAPED:
                        if (nl->err_idev() != 0) {
                Restart:
                                nl->cchar=NLRESTART;
                                nl->cvalid=1;
                                nl->run_odev();
                                nl->istate=IWAIT;
                                nl->icount=0;
                                nl->inext=nl->istart;
                                nullprio(0,0);
                                strtclk();
                                return;
                        }
                        switch (ch) {
                                case NLEOB:
                        Done:
                                        nl->istate=IDONE;
                                        ready(nl->ipid,RESCHNO);
                                        nullprio(0,0);
                                        strtclk();
                                        resched();
                                        return;
```

```
                                case NLSTART:
                                        nl->icount=0;
                                        nl->inext=nl->istart;
                                        return;
                                case NLESC:
                                        if (nl->istate == IESCAPED)
                                                goto Restart;
                                        nl->istate=IESCAPED;
                                        return;
                                default:
                                        if (nl->istate == IESCAPED) {
                                                nl->istate=IREAD;
                                                ch |= NL_UNESC;
                                        }
                                        *nl->inext++  = ch;
                                        if (++nl->icount == nl->imax)
                                                goto Done;
                                }
                                return;
                        case IWAIT:
                                if (ch == NLSTART)
                                        goto SetRead;
                                if (ch == NLEOB)
                                        nl->istate=IREADY;
                                return;
                        case IDONE:
                                return;
                        }
                }
```

Síťová vrstva

- Hlavním úkolem síťové vrstvy v této implementaci je zabezpečení spolehlivosti přenosu po síti. Je zapotřebí navrhnout rutiny síťové vrstvy takovým způsobem, aby se vyšší vrstvy síťového vybavení již nemusely starat o to, zda se nějaký blok dat neztratil nebo zda jednotlivé bloky přicházejí ve správném pořadí. U architektury TCP/IP síťová vrstva zajišťuje nespolehlivé a nespojované služby pomocí protokolu IP.
- Jaksi v ‘nadplánu‘ síťová vrstva zajišťuje rozdělování přijatých *frame* na ty, které je zapotřebí odeslat dále po síti (a ty rovnou předá linkové vrstvě k odeslání), a na ty, jejichž cílovou adresou na úrovni zóny je tento počítač, ty předá vyšším vrstvám síťového vybavení.

- Je třeba navrhnout **protokol** komunikace síťové vrstvy – soustava domluvených pravidel. Součástí protokolu je způsob ohlašování, že při přenosu došlo k chybě, mechanismus vzájemné synchronizace např. po výpadku jednoho z počítačů a podobně.
- Zajištění spolehlivosti - síťová vrstva musí mít k dispozici prostředky, které před odesláním bloku doplní *frame* o kontrolní údaje, a po příjmu porovnáním těchto kontrolních údajů s obsahem ověří, nebyl-li blok během přenosu poškozen. V nejjednodušším případě může být kontrolním údajem pouhá velikost *frame*, porovnaná u příjemce se skutečnou velikostí přijatého bloku (tak tomu např. je v originálním MNU). Pro zvýšení spolehlivosti přenosu však můžeme kontrolní informace rozšířit např. o tzv. checksum nebo kód CRC.

- Je použit mechanismus obecné funkce, která generuje nebo ověřuje kontrolní kódy; implementace této funkce může záviset na konkrétním použití. Funkce by mohla být deklarována např. takto:

```
#include crc.h  /* makecrc, iscrcok */
#define CRC_LEN 2 /* délka kontrolní
                     informace v bytech */
void makecrc(char *buff,unsigned len);
int iscrcok(char *buff,unsigned len);
```

- Funkce `makecrc` uloží na začátek bufferu kontrolní kódy zabírající `CRC_LEN` bytů (předpokládáme, že tam je pro ně vyhrazené místo). Funkce `iscrcok` naopak tyto kódy ověřuje; jsou-li v pořádku (a je-li správná i délka bufferu), vrátí nenulovou hodnotu.
- S využitím funkcí `makecrc` a `iscrcok` tedy síťová vrstva dokáže zjistit, zda je přijatý *frame* v pořádku nebo ne.

- V případě, že je ve *frame* chyba, musí mít síťová vrstva možnost tento fakt oznámit odesílajícímu; pro větší spolehlivost přenosu je navržen protokol tak, že síťová vrstva příjemce bude potvrzovat příjem každého *frame*. Způsob potvrzení se přitom bude samozřejmě lišit v závislosti na tom, byl-li *frame* přijat bez chyby nebo ne - v prvním případě bude potvrzení znamenat ‘mám *frame*, pošli další‘, ve druhém ‘*frame* poškozen, pošli ho znovu‘. Prvnímu případu budeme říkat **pozitivní potvrzení (positive acknowledgement)**, druhému **negativní potvrzení (negative acknowledgement).**
- Mechanismus potvrzování sám o sobě však ještě nezajistí spolehlivý přenos. Představme si velmi jednoduchou situaci, kdy je blok dat přijat bez chyby, ale odpovídající pozitivní potvrzení se ztratí (dojde tedy k chybě při přenosu potvrzení, ne dat). Systém v takovém případě skončí v deadlocku: vysílající bude na věky věků marně čekat na potvrzení odeslaného bloku, zatímco příjemce bude stejně dlouho očekávat další blok.

- Protokol je proto nutné doplnit o **časování (timeout).** Jestliže vysílající ‘nějak příliš dlouho‘ nedostává potvrzení, může se rozhodnout o vlastní iniciativě odeslat blok znovu. Tento mechanismus je velmi důležitý a jestliže nemáme stoprocentně spolehlivé technické vybavení (a takové dosud nikdo nevyrobil a těžko kdy vyrobí), nemůžeme bez něj dosáhnout bezpečného a spolehlivého síťového přenosu.
- Bohužel, ani časování ještě nestačí k dosažení potřebné spolehlivosti. Problém spočívá v tom, že oba propojené počítače pracují zcela paralelně, a proces na jednom z nich proto nemůže nikdy vědět, co právě teď dělá proces na druhém. V našem konkrétním případě to znamená, že ve chvíli kdy síťová vrstva přijímajícího systému odešle potvrzení, mohla by síťová vrstva příjemce omylem potvrzení spojit s jiným blokem. Proto je zapotřebí jednotlivé datové bloky číslovat; součástí potvrzení pak může v každém případě být číslo bloku, který síťová vrstva očekává: číslo chybně přijatého při negativním potvrzení nebo příští číslo při potvrzení pozitivním.

- Existuje důvod, proč není možné číslovat všechny bloky sekvenčně: čísla by v průběhu práce systému příliš narůstala. Nezapomínejme, že potvrzení je kódováno v jediném bytu; aby se do tohoto bytu vešlo i číslo bloku, nesmí zabrat více než několik bitů.
- Po delším výpadku některé ze stanic musíme zajistit **resynchronizaci**, při které se číslování opět 'sejde'. Není proto ani zapotřebí udržovat dlouhou sekvenci čísel bloků, protože po takovéto resynchronizaci by stejně byla porušena.
- Bloky proto budeme číslovat modulo nějaké vhodné malé číslo - běžně se čísla bloků kódují do dvou nebo tříbitových hodnot.
- Číslování bloků a potvrzení však přináší nebezpečí další možnosti zablokování systému - totiž **ztráty synchronizace**. Představme si, že se při přenosu ztratí třeba pozitivní potvrzení bloku číslo 2. Vysílající je vybaven časováním; proto po nějaké době dojde ke správnému názoru, že se potvrzení někde ztratilo a odešle blok číslo 2 znovu. Příjemce však již blok číslo dvě zpracoval; nehodlá proto akceptovat nic jiného, než blok číslo 3. Taková situace by opět mohla trvat na věky věků nebo alespoň do vypnutí počítačové sítě. Je proto zapotřebí protokol doplnit ještě o možnost **resynchronizace.**

- Jestliže příjemce dostává 'nějak příliš dlouho' bloky se špatným pořadovým číslem, odešle vysílajícímu speciální potvrzení vyžadující **resynchronizaci**. Vysílající na základě tohoto potvrzení odešle blok znovu, stejně jako při negativním potvrzení, přidělí mu však číslo 0 (a další bloky čísluje dál od jedničky). Příjemce samozřejmě při odesílání požadavku na **resynchronizaci** inicializoval i své číslování a blok 0 očekává.
- Analyzujeme-li podrobně tento mechanismus vidíme že k zablokování nemůže dojít a že každý *frame* se dostane ke svému příjemci, a to ve správném pořadí. Při ztrátě pozitivního potvrzení však může dojít k tomu, že některý *frame* bude **zduplikován**. Vyšší vrstvy síťového vybavení se tedy ještě musí postarat o vyřazení případných duplicitních dat (což je samozřejmě velmi snadné, pro číslování bloků může použít třeba 32 bitové číslo a duplicitní bloky vyřadí); s touto výjimkou se mohou na síťovou vrstvu plně spolehnout.

- Na první pohled vidíme, že síťová vrstva by se měla skládat ze dvou procesů. První z nich bude číst data ze sítě, ověřovat jejich správnost a odesílat je dál nebo předávat je vyšším vrstvám síťového software. Druhý proces - zcela nezávislý na prvním - bude naopak přebírat datagramy od vyšších vrstev a jako *frame* je bude odesílat dále.
- Pokusíme-li se právě popsaným způsobem síťovou vrstvu skutečně naprogramovat, narazíme na problém s realizací **časování (timeout)** pro odesílající proces. Ten totiž čeká na potvrzení uvnitř služby `getc`; podíváme-li se na její realizaci v linkové vrstvě, uvidíme, že vlastně čeká na přijetí zprávy. Mechanismus zpráv, který máme v XINU k dispozici, neumožňuje přerušit čekání po uplynutí nějakého času.
- Mohli bychom samozřejmě doplnit mechanismus zpráv o novou službu `receive_with_timeout`, která by zajistila právě to, co potřebujeme; obvykle však není vhodné zasahovat při implementaci některé z vrstev operačního systému do nižších a již odladěných vrstev. Jelikož máme k dispozici multitaskingový systém, využijeme jiného mechanismu: vytvoříme třetí proces, který se bude starat o časování sám. Dříve, než zavolá vysílající proces službu `getc` (obsahující volání služby `receive`), aktivuje časovací proces. Ten využije služby `sleep` k tomu, aby nějakou dobu počkal; jestliže odesílající proces dostane potvrzení, opět časovač deaktivuje a nic se neděje. Pokud se však časovač 'dočká' vypršení doby, po kterou měl čekat, pošle prostě volajícímu procesu zprávu `TIMEOUT`. Volající proces zprávu dostane samozřejmě jako návratovou hodnotu služby `getc`; stačí však zvolit kód zprávy `TIMEOUT` odlišný od všech možných potvrzení a vysílající bude moci snadno zjistit, v jaké situaci se nachází.

- Zbývá nám ještě rozhodnout, jakým způsobem budou procesy síťové vrstvy komunikovat s vyššími vrstvami síťového programového vybavení. K tomu se právě velmi dobře hodí mechanismus **portů.** Použijeme dvou portů:
  - na první z nich budou vyšší vrstvy ukládat adresy bufferů s daty, která chtějí odeslat po síti; vysílající proces bude postupně datagramy odebírat a odesílat.
  - na druhý port bude naopak přijímající proces ukládat adresy bufferů, do kterých uložil načtená data; buffery tam budou k dispozici vyšším vrstvám.
- První port bude zároveň sloužit uvnitř síťové vrstvy pro vzájemnou komunikaci přijímajícího a vysílajícího procesu: příjemce na něj bude ukládat adresy bufferů obsahujících *frame*, které je zapotřebí odeslat ihned dále; vysílající je odtamtud bude odebírat a odesílat.

```
/* crc.h - makecrc, iscrcok */

#define CRC_LEN   2     /* délka kontrolní informace v bytech */

void makecrc(char *buff, unsigned len);
int iscrcok(char *buff, unsigned len);
/* end of file */

/* netnet.h */

#define PACK '\x10'       /* pozitivní potvrzení */
#define NACK '\x20'       /* negativní potvrzení */
#define SNACK '\x30'      /* požadavek na resynchronizaci */
#define TIMEOUT 0xff0   /* kód pro timeout */
#define SEQN '\x0f'       /* číslo bloku v potvrzení */

#define BCAST 0xff        /* rozeslání bloku všem počítačům */

#define ACKTMO     4      /* timeout při ztrátě potvrzení */
#define SFAIL      2      /* počet chyb sekvence pro SNACK */
#define SMODULO  7        /* modulo pro sekvenční čísla */

#define FMINLEN  CRC_LEN+5          /* minimální velikost frame */
#define FDATALEN   128  /* max. velikost datagramu */
#define FLEN sizeof(struct frame)   /* max. velikost frame */
```

```
struct frame {
        char ctrl[CRC LEN];             /* kontrolní údaje */
        char from,to;                   /* odesilatel a adresát frame */
        char seq;                       /* sekvenční číslo */
        int len;                        /* délka datagramu (<=FDATALEN) */
        char data[FDATALEN];            /* data frame, tj. datagram */
}

struct frnet {        /* globální data síťové vrstvy */
        int iport;              /* port pro načtené bloky */
        int oport;              /* port pro odesílaná data */
        int idev;               /* číslo vstup. síťového zařízení */
        int odev;               /* číslo výst. síťového zařízení */
        int timer;              /* čítač pro timeout */
        int tmpid;              /* časovací proces */
        int tdpid;              /* proces, který čeká na timeout */
        char stid;              /* síťové číslo tohoto počítače */
        char iseq;              /* čítač sekvence pro vstup */
        char sfails;            /* počet chyb sekvence při vstupu */
        char oseq;              /* čítač sekvence pro výstup */
};

extern struct frnet frnets[];
```

- Význam všech polí je jasný z komentářů; zvláštní vysvětlení si snad zaslouží jen symbolická konstanta `BCAST`:
  - někdy může být vhodné určitý paket rozeslat všem počítačům na celé síti (může se to hodit třeba pro rozesíláni administrativních údajů jako je změna konfigurace sítě). Budeme tedy chtít, aby síťová vrstva odpovídající *frame* předala vyšším vrstvám programového vybavení a zároveň jej odeslala dál jako cizí *frame*. To samozřejmě není nic těžkého musíme jen síťové vrstvě tento požadavek nějak oznámit. K tomu právě slouží hodnota *BCAST (broadcast)*, kterou ve zmíněném případě použijeme na místě síťové adresy cílového počítače.
- 'Globální' proměnné pro síťovou vrstvu jsou uloženy v tabulce `frnets` proto, aby mohl jeden počítač obsluhovat více zón sítě. V tomto smyslu budeme také nadále používat pojem 'číslo zóny': jako interní číslo zóny v rámci jednoho počítače, čili jako jeho index do pole `frnets`.

Rozhraní pro vyšší vrstvy

```
/* freceive.c — freceive */
#include <conf.h>
#include <crc.h>
#include <net_net . h>

/*
* freceive — čtení frame ze sítě
*/

char *freceive(int net)
{
      return(preceive(frnets[net] .iport));
}
/* end of file */
```

```
 /* fsend.c — fsend */
#include <conf.h>
#include <crc.h>
#include <net_net . h>

/*
* fsend - odeslání frame po síti
*/

void fsend(int net, char wkid, char *buff)
{
      struct frnet *fn = frnets[net];
      struct frame *fm = buff;

      fm->to = wkid;
      fm->from = fn->stid;
      psend( fn->oport , buff);
}
/* end of file */
```

Vstupní proces

- Vstupní proces používá pro odesílání potvrzení pomocné služby `sendack`, `sendnack` a `sendsack`. První z nich odešle pozitivní potvrzení, druhá negativní potvrzení a třetí podle počtu sekvenčních chyb odešle buď negativní potvrzení nebo požadavek na resynchronizaci.
- Nejprve proces inicializuje lokální proměnné, alokuje blok paměti pro načtený *frame* a pak přejde do nekonečného cyklu. To je běžný případ systémových procesů - celou dobu tráví v nekonečném cyklu, ve kterém zpracovávají potřebná data. Většinu doby samozřejmě takový proces nepracuje, ale v rámci některé ze systémových služeb čeká mimo ready frontu (zde je 'čekací' službou služba `read`, která - jak jsme viděli v popisu linkové vrstvy - přemístí proces do stavu *suspended*.

- První, co proces v cyklu udělá, je čtení bloku dat prostřednictvím ovladače síťového zařízení (jinými slovy prostřednictvím linkové vrstvy). Po přijetí bloku proces nejprve ověří, může-li se vůbec jednat o *frame*, tj. je-li blok alespoň tak dlouhý, aby se do něj vešla hlavička *frame*. Pak si vyžádá ověření dat *frame* službou `iscrcok`. Jestliže blok není v pořádku, odešle negativní potvrzení a čte další blok.
- Byl-li blok v pořádku, srovná proces jeho sekvenční číslo s očekávaným sekvenčním číslem. Jestliže se obě čísla liší, odešle proces prostřednictvím služby `sendsack` buď negativní potvrzení nebo požadavek na synchronizaci a opět čte další blok
- V případě, že byl blok v pořádku a měl správné sekvenční číslo, přejde proces na zpracování *frame*.
  - První příkaz `if` zjistí, není-li příjemcem `BCAST` - nejedná-li se tedy o zprávu, rozesílanou po celé síti. Je-li tomu tak, předá *frame* v každém případě relační vrstvě; pokud byl *frame* vytvořen na jiném počítači, odešle jej také dále.
  - Smysl tohoto mechanismu je jasný: jeden počítač zprávu typu `BCAST` vygeneruje. Zpráva projde všemi ostatními počítači v kruhové síti XINU; každý počítač si ji ponechá a zároveň ji odešle dále. Když se zpráva dostane opět na počítač, který ji odeslal původně, nebude se již samozřejmě posílat znovu dokola; vyšší vrstva programového vybavení ji však dostane jako potvrzení, že zpráva prošla skutečně celou sítí.
  - Druhý příkaz `if` ověří, není-li lokální počítač příjemcem *frame*. Je-li tomu tak, předá *frame* relační vrstvě.
  - Třetí příkaz `if` srovná odesílatele *frame* s číslem lokálního počítače. Pokud se shodují, znamená to, že příjemce z nějakého důvodu *frame* neodebral. Proces tuto situaci oznámí uživateli a *frame* 'zahodí' - tj. nechá do téže paměti číst další blok (proto `goto SkipAlloc`).
  - Setkáváme se zde se speciální službou `syslog`. Ta nedělá nic jiného, než že vypíše své argumenty 'systémového záznamu'. Tím může být třeba speciální soubor nebo zvolené okno na obrazovce, nebo tiskárna připojená k počítači - to záleží na konkrétní implementaci systému. Operační systém službu `syslog` používá pro hlášení všech důležitých nebo podivných situací; hledá-li pak administrátor systému řešení nějakého problému, může mu být systémový záznam velmi významným pomocníkem. Systémový záznam (spojený s intenzívním používáním služby `syslog`) je také prakticky nutnou podmínkou pro odladění operačního systému.
  - Jestliže *frame* 'nepoznal' žádný z příkazů `if`, jedná se o cizí *frame* a proces jej prostě odešle dále (`goto Sendlt`).
  - Paměť, ve které je přijatý blok uložen, není ještě možné uvolnit (s výjimkou případu třetího `if`) - data v ní čekají buď na odeslání na další počítač, nebo na zpracování relační vrstvou. Proces proto musí alokovat další blok paměti službou `new`.
  - Potom proces již jen odešle pozitivní potvrzení a čte další blok ze sítě.

```
/* sendack  / odešle pozitivní potvrzení se sekvenčním číslem */

void sendack(struct frnet *fn)
{
        fn->sfails = 0;
        if ( ++fn->iseq > SMODULO)
                fn->iseq = 0;
        putc(fn->idev, PACK | fn->iseq);
}

/* sendnack —- odešle negativní potvrzení se sekvenčním číslem  */
void sendnack(struct frnet *fn)
{
         putc(fn->idev, NACK | fn->iseq);
}

/*  sendsack -- zpracuje situaci po chybě sekvence */
void sendsack(struct frnet *fn)
{
        if (++fn->sfails == SFAIL) {
                fn->iseq = 0;
                fn->sfails = 0;
                putc( fn->idev, SNACK);
        } else
                putc ( fn->idev , NACK);
}
```

```
/* finpproc —— proces */

void finpproc(int net)
{
        struct frnet *fn = frnets[net];
        struct frame *fm;

        new(fm);
        fn->iseq = 0;
        fn->sfails = 0;
        for (;;) {    /* forever */
                int len;
                if ( ( len=read(fn->idev, fm, FLEN) ) < FMINLEN || !iscrcok(fm,len) )
                        sendnack(fn);
                else if (fm->seq != fn->iseq)
                        sendsack(fn);
```

```
        else {
                if (fm->to == BCAST) {
                        if ( fm->from == fn->stid)
                                psend( fn->iport, fm);
                        else {
                                struct frame *f;
                                new( f);
                                bcpy(f,fm,len);
                                psend( fn->iport , f);
                        SendIt:
                                psend(fn->oport, fm);
                        }
                } else if (fm->to == fn->stid)
                         psend( fn->iport, fm);
                else if (fm->from = fn->stid) {
                        syslog("nemám spojení s %d ?!?",fm->to);
                        goto skipAlloc;
                } else goto sendIt;
                new(fm);
        SkipAlloc:
                sendack(fn);
                }
        }
}
```

Výstupní proces

- Výstupní proces používá podobně jako vstupní také svou pomocnou službu - její název je `frsend`. Tentokrát je však tato pomocná služba daleko složitější než samotný proces - ten totiž pouze odebírá jednotlivé *frame* z výstupního portu a předává je právě službě `frsend` k odeslání.
- Služba `frsend` tedy musí zajistit nejen odeslání *frame* - což je jednoduché, stačí předat *frame* linkové vrstvě - ale i zpracování potvrzení a případnou resynchronizaci.
- Služba nejprve připraví kontrolní údaje pomocí funkce `makecrc`, pak zjistí právě platné sekvenční číslo bloku pro výstup a vstoupí do cyklu `for`. Tentokrát se nejedná o věčný cyklus, jako v démonech - cyklus bude ukončen po přijetí pozitivního potvrzení se správným sekvenčním číslem.
- Funkce nejprve inicializuje příjem potvrzení službou `getc` linkové vrstvy, a potom teprve skutečně odešle *frame* službou `write`. Pak nastaví čítač časovacího procesu na hodnotu `ACKTMO`, časovací proces aktivuje a čeká na potvrzení pomocí služby `receive`. Ta může vrátit některou z následujících hodnot:

- `PACK` s číslem příštího požadovaného bloku, jestliže druhý počítač odeslaný blok bez problémů přijal;
- `NACK`, jestliže druhý počítač blok přijal chybně;
- `SNACK`, jestliže druhý počítač požaduje resynchronizaci;
- `TIMEOUT`, jestliže od druhého počítače nepřišlo vůbec žádné potvrzení tak dlouho, že mezitím časovací proces vyčerpal čítač `timer`.

- Ihned po návratu ze služby `receive` proto funkce `frsend` nejprve deaktivuje časovací proces uložením záporné hodnoty do čítače, a pak rozliší potřebné alternativy pomocí příkazu `switch`:

- Varianty NACK a TIMEOUT není zapotřebí vůbec detekovat - cyklus automaticky zajistí nové odeslání bloku, což je přesně to, co v takovémto případě chceme dělat. Abychom dali příjemci nějaký čas na zotavení (a abychom nechali lokálnímu počítači také trochu času pro práci, jestliže druhý počítač odmítá vůbec komunikovat), zavoláme službu sleep.
- Po přijetí potvrzení `PACK` ještě zkontrolujeme, je-li v pořádku sekvenční číslo bloku. Jestliže ano, byl blok skutečně bez problémů přijat a služba `frsend` může skončit. Jestliže ne, odešleme blok znovu.
- Konečně po přijetí požadavku `SNACK` nejprve synchronizujeme sekvenční číslo na nulu, pak ponecháme příjemci (který samozřejmě také musí synchronizovat) ještě více času než obvykle voláním služby `sleep`, a blok odešleme znovu.

```
/* frsend —— odeslání frame, zpracování potvrzení */

void frsend(struct frnet *fn, struct frame *fm)
{
        char seq,nextseq;
        int len;
        int msg;

        fm->seq = seq = fn->oseq;
        nextseq = seq  == SMODULO ? 0 : seq+1;
        makecrc ( fm, len = fm->len+FMINLEN);
         for (;;) {
                getc ( fn->odev) ;
                write ( fn->odev, fm, len);
                fn->timer = ACKTMO;
                fn->tdpid = currpid;
                resume ( fn->tmpid );
                msg = receive();
                fn->timer = -1;
```

```
                switch (msg & ~SEQN) {
                        case PACK:
                                if (msg & SEQN == nextseq) {
                                        fn->oseq = nextseq;
                                        return;
                                }
                                break;
                        case SNACK:
                                fm->seq = fn->oseq = 0;
                                nextseq = 1;
                                sleep(1);
                }
                sleep(1);
        }
}
```

```
/*  foutproc.c — výstupní proces síťové vrstvy */

#include <conf.h>
#include <crc.h>
#include <net_net . h>

/* foutproc —— proces */

void foutproc(int net)
{
      struct frnet *fn = frnets[net];
      struct frame *fm;

      fn->oseq=0;
      for (;;) {        /* forever */
            frsend ( fn , fm = preceive ( fn->oport ));
            dispose(fm);
      }
}
```

Časovací proces

- Časovací proces je velmi jednoduchý. Jeho jediným úkolem je počítat čas na základě čítače `timer`, a jestliže dojde k nule, poslat procesu `tdpid` zprávu `TIMEOUT`. Z technických důvodů je proces navržen tak, aby bylo možné jej ‘zastavit‘ uložením záporné hodnoty do čítače; to je proto, že pokud by náhodou proces zrovna čekal ve stavu `sleeping`, nelze jej přímo suspendovat.
- Při inicializaci využijeme toho, že všechny procesy jsou automaticky službou `create` vytvořeny ve stavu `suspended`. Procesy pro vstup a pro výstup při inicializaci ihned převedeme službou `resume` do ready fronty; Časovací proces však ponecháme suspendovaný - je navržen tak, že bude stejně dobře pracovat po aktivaci na samém začátku, jako po aktivaci uvnitř jeho služby `suspend`.

```
/* ftimeproc.c — časovací proces síťové vrstvy */

#include <conf.h>
#include <crc.h>
#include <net_net . h>

/* ftimeproc proces */

void ftimeproc(int net)
{
      struct frnet *fn = frnets[net];

      for (;;) {        /* forever */
            for (fn->timer++; --fn->timer>0;)
                  sleep(10);
            if (fn->timer == 0) send(fn->tdpid, TIMEOUT);
            suspend(currpid);
      }
}
```

Řízení přístupu

Architektura autentizace, autorizace a účtováni (authentication, authorization and accounting, AAA) používaná pro ochranu počítačových sítí z hlediska přístupu, v sobě skrývá:

AUTENTIZACI

Autentizace je proces identifikace a verifikace uživatele. Uživatele lze identifikovat třemi možnými způsoby, jichž se využívá pro kontrolu oprávněnosti jejich přístupu k síti a ověření jejich práv vykonávat určité úkony, podle toho:

- **kdo jsou** - identifikace podle globálně jednoznačných ukazatelů (otisky prstů, dlaní apod.).
  + nelze zaměnit.
  - zařízení pro identifikaci jsou velmi nákladná,
- **co mají** - identifikace podle vlastnictví určitých předmětů (klíčů, magnetických karet apod.)
  + jednodušší možnost ověření autorizace,
  - náchylné ke ztrátám, kopiím, krádežím.
- **co vědí** - identifikace pomocí přístupových hesel, číselných kombinací, osobních identifikačních čísel apod.
  + nejjednodušší způsob zabezpečení,
  - náchylné k zapomenutí, zneužití v případě jejich záznamu na jakémkoli médiu, vyžadují pečlivý výběr (sestavení) pro minimalizaci uhádnutí, a někdy pravidelnou obměnu.

AUTORIZACI a **ÚČTOVÁNÍ**

Po úspěšné autentizaci uživatele může být udělena autorizace pro užívání síťových zdrojů a služeb. Autorizace specifikuje, co uživatelé mohou v systému provádět za operace a jaká data jsou pro ně dostupná.

Poslední složka architektury ochrany systému je účtování, které zodpovídá za záznam všech činností uživatele v systému. V případě problémů je možné ze záznamů dohledat viníka.

Systém služeb

- Operační systém musí zajistil maximální množství často požadovaných služeb tak, aby každý programátor nemusel vytvářet pokaždé znovu a znovu stejný kód.
- Jak se postupně vyvíjely operační systémy, objevovalo se samozřejmě čím dál tím více takových služeb. I když nejdůležitější částí každého moderního operačního systému zůstává správa prostředků, bývají co do objemu služby daleko a daleko rozsáhlejší.
- Kromě vlastního množství služeb je samozřejmě nesmírně důležitá i jejich univerzálnost a možnost jejich mírného přizpůsobení konkrétním požadavkům právě vyvíjeného programu tak, aby programátor byl nucen vytvářet skutečně jen a pouze nový kód.

Jaké služby potřebujeme

- Od samého začátku samozřejmě všechny operační systémy nabízely programátorům alespoň základní služby pro přidělování a uvolňování operační paměti (nemáme-li k dispozici virtuální adresový prostor, ani by to bez nich nešlo) a poměrně komfortní služby pro práci se soubory.
- Prostřednictvím ovladačů zařízení, připojených k systému, měli programátoři navíc k dispozici i služby pro práci s těmito zařízeními; málokdy se však jednalo o služby dostatečně vysoké úrovně.
- Multitaskové operační systémy pak samozřejmě musí nabízet služby správce procesů.
- Dnes to ani zdaleka nestačí. Operační systém musí nabízet poměrně velmi luxusní služby alespoň v následujících oblastech:

- **Grafické operace**, práce s okny a graficky orientovaná komunikace s uživatelem, např. volba stylu písma, volba barvy, editace textu v rozsahu jediného řádku i na úrovni kompletního editoru a podobně. Do této oblasti patří také tisk (v rozumném operačním systému je tisk textu pouze speciálním případem tisku grafiky
- **Práce s textovými řetězci a s bloky dat** v operační paměti. Nejrůznější přesunování textů nebo skupin bytů, vyhledávání v nich a jejich kombinace patří snad mezi nejčastější základní operace, které kterýkoli program provádí. Zpracování textu pak je základní součástí prakticky všech programů, které vůbec nějak komunikují s uživatelem.
- Tato problematika velmi úzce souvisí s graficky orientovanou komunikací s uživatelem - je totiž nutné si uvědomit, že práce s textem nekončí u knihoven jazyka C a standardních služeb pro kopírování či připojení textového řetězce; služby moderního operačního systému musí být schopné pracovat s **formátovaným textem** obsahujícím údaje o použitém fontu, velikosti a typu písma, o zarovnání jednotlivých odstavců, o použité barvě, …
- Není-li operační systém určen výhradně pro procesory vybavené jednotkou pro výpočty v pohyblivé řádové čárce (nebo alespoň odpovídajícím koprocesorem), je zapotřebí, aby obsahoval rozsáhlou skupinu **služeb pro práci s neceločíselnými hodnotami.**
- Téměř pro každé zařízení (jako příklad jmenujme zvukový vstup a výstup nebo systémové hodiny) by měl operační systém nabízet skupinu služeb, které samy zpracují nejběžnější požadavky kladené na zařízení. S našimi příklady by tedy systém jistě neměl ponechávat na každém programátorovi, aby se sám starat o generování tónů požadované výšky, intenzity, délky a barvy nebo o přehrávání samplovaného zvukového záznamu; analogicky pro zvukový vstup by měl systém zajistit dekódování a případné uložení samplovaného záznamu.

- Nesmírně důležitou oblastí je podpora **práce v cizím jazyce**. Operační systém by měl nabízet řadu prostředků umožňujících jak práci s dokumenty psanými v jiném jazyce, než pro který byl systém původně navržen, tak i vlastní komunikaci s uživatelem v ‘jeho‘ jazyce. Nejmodernější operační systémy pak mohou s každým ze svých uživatelů komunikovat jinou řečí podle jeho osobní volby.
- Jednou z nejčastějších problematik řešených na počítačích je ukládání a opětovné vyhledávání údajů - tedy databáze. Dokonce i řada ‘nedatabázových‘ úloh se dá vyřešit daleko pohodlnějším a efektivnějším způsobem. je-li možné jako základ řešení použít již hotový, fungující a rychlý databázový systém. Programátoři by proto v moderním operačním systému měli mít k dispozici i poměrně velmi rozsáhlý balík **databázových služeb**.
- Velmi významnou oblastí je i **komunikace mezi programy** na vyšší úrovni. Jednotlivé procesy samozřejmě mohou snadno komunikovat s využitím aparátu zpráv nebo semaforů, které jím nabízí správce procesů; je však zapotřebí umožnit aplikacím, aby si mohly bez extrémního úsilí a explicitní dohody jejich programátorů navzájem předávat data ke zpracování, informace o stavu těchto dat, jejich případné modifikaci a podobně.

Implementace služeb

- V klasických operačních systémech existovaly v zásadě dvě možnosti implementace služeb. Buď se mohlo jednat o plnohodnotné **systémové služby**, které operační systém zajišťoval na vyžádání, nebo mohly být služby uloženy na **knihovnách**, odkud se při vytváření aplikačních programů připojily k jejich kódu.
- Dnešní operační systémy obvykle disponují dvěma dalšími prostředky pro implementaci služeb. Jedná se o **servery** a o **sdílené knihovny**. Nejmodernější objektově orientované operační systémy pak mají k dispozici navíc také **aparát komunikace objektů**.
- V praxi operační systém pro zajištění služeb obvykle kombinuje všechny popsané metody.

Systémové služby

- **Systémové služby** jsou obvykle vyvolávány pomocí speciální instrukce procesoru, která přepne pracovní režim z uživatelského do systémového a předá řízení předem zvolené rutině operačního systému. Ta na základě vstupních parametrů zjistí, kterou službu program požadoval a předá řízení rutině, která služby zpracuje (viz např. dispečer služeb ovladačů). Nejčastěji se používají instrukce pro vyvolání programového přerušení; řada procesorů má navíc některé speciální instrukce pro tyto účely.
- Parametry se při vyvolání systémových služeb nejčastěji předávají v **registrech**. To je výhoda z hlediska rychlosti zpracování služby; je to však nepohodlné v případě. kdy potřebujeme předávat větší množství formátovaných parametrů a musíme je nějak umístit do několika ‘bezformátových‘ registrů. Systémové služby proto v praxi bývají velmi často ‘zabaleny‘ do velmi jednoduchých služeb (umístěných na klasických knihovnách), které pouze překódují parametry z ‘pohodlného‘ tvaru uloženého nejčastěji na zásobníku do registrů ž pak zavolají odpovídající systémovou službu.

- Další a nejvýznamnější nevýhodou **systémových služeb** je právě to, že jsou **přímo součástí operačního systému**. Připomeňme si obrovský rozsah služeb - takový operační systém by se sám nevešel do paměti (pokud bychom neměli k dispozici virtuální paměť). Navíc by v obrovském operačním systému - i přesto, že by byl samozřejmě složen a řady modulů - přece jen narůstalo riziko chyb.
- Hlavní výhodou systémových služeb je jejich ‘**právo dělat cokoli**‘. Jsou-li služby součástí systému, je také jejich kód zpracováván v systémovém režimu procesoru; mohou tedy přímo ovládat jednotlivá zařízení nebo kanály, mohou maskovat jednotlivá přerušení, mohou přeprogramovávat jednotku řízení paměti a podobně. Systémové služby proto musí v každém případě pokrýt všechny úlohy, pro které je provádění takovýchto potenciálně nebezpečných akcí nezbytné; ostatní úkoly zpravidla daleko vyšší úrovně - pak mohou být realizovány jinak.

Klasické knihovny

- Princip klasických knihoven je asi znám všem programátorům. Program se obvykle vytváří ve dvou krocích:
  - Nejprve se pomocí překladačů vytvoří jednotlivé moduly. V modulech samozřejmě je množství volání nejrůznějších služeb; na každém takovém místě je instrukce volání podprogramu bez cílové adresy, namísto této adresy je v modulu uloženo jméno požadované služby. Modul může nějakou službu také sám nabízet; pak je jeho součástí informace o jménu služby a o relativní adrese odpovídajícího podprogramu uvnitř modulu.
  - Druhým krokem je spojení všech modulů dohromady. Spojovací program (linker) přitom má k dispozici nejen moduly. které vytvořil programátor aplikace, ale i množství modulů, které jsou uloženy právě v systémových knihovnách. Program pak vybere všechny moduly z knihoven, které obsahují služby volané z ‘aplikačních‘ modulů, a ze všech modulů dohromady vytvoří hotový program. Všechny instrukce volání podprogramů přitom doplní správnými adresami.
- Ve výsledném programu je tedy uložen kompletní kód služeb z knihoven, stejně, jako by jej programátor zapsal při tvorbě programu.
- To je hlavní **nevýhodou klasických knihoven**. Máme-li totiž potom sto programů používajících služby z klasických knihoven na disku, máme na disku sto jedna kopií téhož kódu (sto v programech, sto prvá kopie je uvnitř knihovny, která je na disku obvykle uložena také). Zavedeme-li v multitaskovém prostředí třicet takovýchto programů do paměti, budeme v ní mít opět třicet kopií téhož kódu.

- Žádnou významnější výhodu klasické knihovny nemají (aparát sdílených knihoven - není-li pro něj využit systém virtualizace adres - může znamenat určité zvýšení režie; není to však obvyklé). Asi hlavním důvodem, proč klasické knihovny dosud ani v moderních operačních systémech nevymizely, je to, že jsme na jejich používání zvyklí.
- V dnešních operačních systémech klasické knihovny obvykle obsahují pouze kratičké pomocné funkce, které převedou normální volání podprogramu na vyvolání systémové služby nebo na odeslání zprávy serveru.
- Klasické knihovny mohou sloužit i pro dosažení kompatibility se staršími systémy nebo pro usnadnění práce programátorům, kteří jsou zvyklí na klasické prostředky a nechtějí přecházet na objektově orientované systémy.

Servery

- Se **servery** (speciálními procesy, nabízejícími ostatním určité služby) jsme se již několikrát setkali. Stačí si uvědomit, že server nemusí být určen pouze pro obsluhu nějakého fyzického zařízení, ale může také naprosto stejným způsobem nabízet služby, ovládající nějaké ‘zařízení‘ logické - např. systém souborů nebo databází.
- Výhody a nevýhody serverů se do jisté míry podobají výhodám a nevýhodám služeb samotného operačního systému. Hlavní rozdíl spočívá v tom, že servery obvykle nepracují v systémovém režimu procesoru (a musí tedy samy využívat systémových služeb pro přímé ovládání zařízení). Zůstává společná nevýhoda zabrané paměti (není-li k dispozici virtuální paměť), i nevýhoda snadnějšího zavedení chyb do velkého programu, jakým takový server obvykle bývá. Vzhledem k tomu, že pro každý úkol může být určen samostatný server, však tato nevýhoda není ani zdaleka tak markantní, jako tomu bylo v případě systémových služeb.
- Servery jsou relativně ‘samostatné‘; kód z klasických knihoven je naproti tomu součástí programu, který jej využívá. Servery proto častěji slouží tam, kde je zapotřebí provádět nejen akce na přímou výzvu programu. ale i akce asynchronní, ‘o vlastní vůli‘ - dobrým příkladem může být třeba správa souborů (s asynchronní obsluhou cache pamětí) nebo inteligentní ovladač obrazovky (který požadovaný výstup dokresluje také asynchronně).

Sdílené knihovny

- Udržování tolika kopií jediného kusu kódu při použití klasických knihoven je nešikovné a lze to vyřešit lépe použitím tzv. **sdílených knihoven**. Jejich funkce je podobná jako funkce klasických knihoven, právě jen s jediným podstatným rozdílem: není zapotřebí, aby na disku nebo v paměti byly více než jednou (máme-li k dispozici systém virtuální paměti, stačí jediná kopie pro disk i pro paměť dohromady).
- Princip spočívá v tom, že spojovací program svou práci ‘nedodělá‘ a ponechá to na **zavaděči**, který ukládá programy při spuštění do operační paměti. Celý mechanismus pak vypadá přibližně takto:

  – Nejprve se pomocí překladačů vytvoří jednotlivé moduly, naprosto stejně jako tomu bylo v případě knihoven klasických.
  – Druhým krokem je opět spojení všech modulů dohromady. Spojovací program však tentokrát vyřeší pouze vzájemné odkazy mezi moduly, které vytvořil programátor aplikace, sdílených knihoven si nevšímá a odkazy na služby, které jsou na sdílených knihovnách uloženy, ponechá beze změny.
  – Teprve při zavádění programu do paměti zavaděč zjistí, které ze sdílených knihoven jsou zapotřebí. Pak ověří, nejsou-li již tyto knihovny v paměti (jako důsledek zavedení některého z minulých programů); jestliže tomu tak není, do paměti je zavede.
  – Potom teprve zavaděč uloží do paměti i kód spouštěného programu; přitom jeho instrukce pro volání podprogramů ze sdílených knihoven doplní správnými adresami, odvozenými od umístění sdílené knihovny v paměti.

- Při používání sdílených knihoven vyplývá navíc jedna podstatná výhoda, která přináší do systémů se sdílenými knihovnami určitý rys systémů objektových:
  - jestliže totiž nahradíme starou verzi operačního systému verzí novější, budou všechny programy - i ty, které byly dohotoveny v době, kdy se nikomu ještě o nové verzi systému nesnilo - automaticky při zavádění spojovány s novými knihovnami a budou tedy automaticky využívat jejich výhod (odstraněných chyb, vylepšených algoritmů, doplněných nových funkcí a podobně).
- Samozřejmě nové knihovny musí zachovat stejné rozhraní pro implementované funkce (počet parametrů, význam parametrů, apod.)
- Pokud chceme použít nové funkce, budeme muset aplikační program samozřejmě upravit a znovu přeložit.

Obsah služeb

- **Textové služby** můžeme velmi snadno rozdělit do dvou skupin.
  - V první z nich jsou jednoduché služby pro nejrůznější přesuny dat v operační paměti; nejedná se jen o pohodlí programátora - operační systém může často přesuny větších bloků dat zajistit rychleji než aplikační program, protože mívá k dispozici speciální vybavení - může např. využít tzv. DMA kanál, může selektivně vypínat cache paměť a podobně).
  - Druhá skupina je komplikovanější: jde o zpracování formátovaných textů zahrnujících různé fonty, velikosti písma, barvy a podobně. Je nutné, aby sám operační systém práci s formátovaným textem podporoval, a to ze dvou důvodů:
    - Práce s formátovaným textem je příliš častým případem, než aby se vyplatilo ji programovat pokaždé znovu.
    - Druhým a hlavním důvodem je komunikace programů. Jde o to, aby bylo možné bez problémů přenášet formátovaný text mezi dvěma aplikacemi, jejichž tvůrci se na ničem nedomluvili.

- Služby pro práci s **formátovaným textem** (i některé služby pro práci s textem neformátovaným) navíc velmi úzce souvisejí s **podporou národního prostředí**. Součástí mnoha služeb je totiž také interpretace jednotlivých znaků (např. při převodu malých písmen na velká nebo při dělení odstavce na řádky). Operační systém musí v takovém případě korektně bez extrémního úsilí programátora konkrétní aplikace zajistit správnou interpretaci národních znaků.

Národní prostředí

- Služby pro práci s formátovaným textem (i některé služby pro práci s textem neformátovaným) navíc velmi úzce souvisejí s podporou **národního prostředí**. Součástí mnoha služeb je totiž také interpretace jednotlivých znaků (např. při převodu malých písmen na velká nebo při dělení odstavce na řádky). Operační systém musí v takovém případě korektně bez extrémního úsilí programátora konkrétní aplikace zajistit správnou interpretaci národních znaků.
- I podpora národního prostředí se dá snadno rozdělit na dvě skupiny služeb:
  - první skupina umožňuje uživateli počítače zpracovávat dokumenty v různých jazycích
  - druhá skupina služeb pak je určena k tomu, aby uživatel mohl s počítačem komunikovat ve své rodné řeči.
- Obě skupiny jsou velmi komplexní; mezi hlavní součásti prvé skupiny patří:
  - Možnost zobrazovat speciální znaky, potřebné pro požadovaný jazyk, na obrazovce počítače. Je známo, že to není zcela triviální ani s českými diakritickými znaménky; skutečné problémy však nastávají s východními abecedami, jejichž znaky jsou velmi komplikované a bývá jich značný počet (např. čínské a japonské abecedy).

- Neméně důležitá je i možnost přenést texty v požadovaném jazyce na všechna potřebná výstupní zařízení - může se jednat o tiskárnu, fax, osvitovou jednotku nebo třeba stroj na výrobu kreditních karet. Zde je velkou výhodou grafický subsystém se službami vysoké úrovně, který umožňuje využití jediného mechanismu zobrazování pro kterékoli výstupní zařízení. Pak stačí vyřešit zobrazování národních znaků jen jednou a není nutné se jím zabývat pro každé nové výstupní zařízení vždy znovu a znovu.
- Nemalým problémem je i vhodné ovládání klávesnice. Řada jazyků má příliš mnoho znaků, než aby bylo možné je rozumným způsobem poskládat na běžnou počítačovou klávesnici. Je proto nutné zavádět tzv. **mrtvé klávesy**, různé pracovní režimy a další speciální prostředky pro vstup znaků.
- Klasifikaci a převody znaků. Mnoho programátorů v jazyce C např. používá často standardní službu 'isupper', která ověří, je-li zadaný znak velkým písmenem; málokdo si však přitom uvědomí, že pokud tato služba nepozná také velká písmena 'A', 'Č', 'Ď', ..., není její implementace korektní.
- Totéž platí pro převody malých a velkých písmen. Standardní systémové služby musí převádět korektně všechny znaky, včetně národních, a musí být automaticky přístupné všem programátorům prostřednictvím standardních funkcí (jakými jsou např. funkce 'toupper' nebo 'tolower' v jazyce C).

- Řadu stále složitějších a složitějších problémů vyvolává tak základní úkol, jakým je setřídění několika slov nebo vět. Operační systém samozřejmě musí třídění zajišťovat sám v závislosti na aktivním jazyce; při implementaci třídicích rutin je však zapotřebí:
- Nejprve zajistit, aby vůbec všechna písmena měla správné pořadí, pro češtinu tedy např. musí být 'á' před 'b', ačkoli má ve všech běžně užívaných kódováních vyšší kód.
- Pak narazíme na písmena, složená z dvojice znaků, která se však z hlediska třídění chovají skutečně jako jediné písmeno (například v češtině máme 'ch').
- Po vyřešení obou problémů zjistíme, že třídění v některých jazycích - mezi které čeština patří- dosud není korektní. Správné třídění v nich totiž z jakýchsi prapodivných a nepochopitelných důvodů není lexikografické, ale daleko složitější – např. správné pořadí českých slov 'Děkan', 'Dentista' a 'Děvín' je to, ve kterém jsme je uvedli, ačkoli první a třetí slovo začínají stejně.
- Vyřešíme-li přece Jen všechny problémy a implementujeme korektní třídění, musíme vyřešit také problém, kde jej použít a kde ne. Má být např. seznam souborů setříděn lexikograficky nebo podle aktivního jazyka?
- Součástí podpory národního prostředí je i 'překlad' automaticky generovaných údajů (jako jsou data, jména dní a měsíců apod.). Často je překlad velmi obtížný, protože v některých jazycích - mezi nimiž samozřejmě čeština patří - se slova ohýbají, takže jedno slovo může v různých kontextech vypadat jinak (březen – března) .

- Komplexní textové služby na národním prostřední závisejí také velmi podstatným způsobem - množství jazyků standardně píše zprava doleva a některé píší dokonce ve sloupcích'. Služby operačního systému by samozřejmě měly podporovat i toto.
- Nikoli nepodstatnými službami systému je i podpora pro dělení slov na konci řádků a korektor (spellchecker).

- Druhou skupinou služeb, které musí operační systém zajistit, je podpora komunikace s uživatelem ve zvoleném jazyce. Zásadním problémem je pochopitelně vlastní překlad textů. Přitom však je nutné si uvědomit, že je často zapotřebí 'přeložit' i obrázky, které mohou mít výrazně odlišný význam v jiné kulturní oblasti. Černobíle zobrazená pěticípá hvězda např. v českém uživateli vyvolá zcela jiné asociace a představy než v Američanovi; jiným příkladem je třeba číslo 69, které je u nás prakticky bez emočního náboje, zatímco v USA je považováno za téměř vulgární.
- Operační systém musí obsahovat podporu pro práci **vícejazyčných aplikací**. Vícejazyčná aplikace je program, který obsahuje komunikační prvky pro několik různých jazyků. Teprve ve chvíli spuštění se ve spolupráci s operačním systémem automaticky zvolí jeden z těchto jazyků - ten, který uživateli nejlépe vyhovuje - a kdykoli pak program použije některý z komunikačních prvků, zajistí systémové služby automaticky použití prvku z tohoto jazyka.

Poznámky k systému UNIX

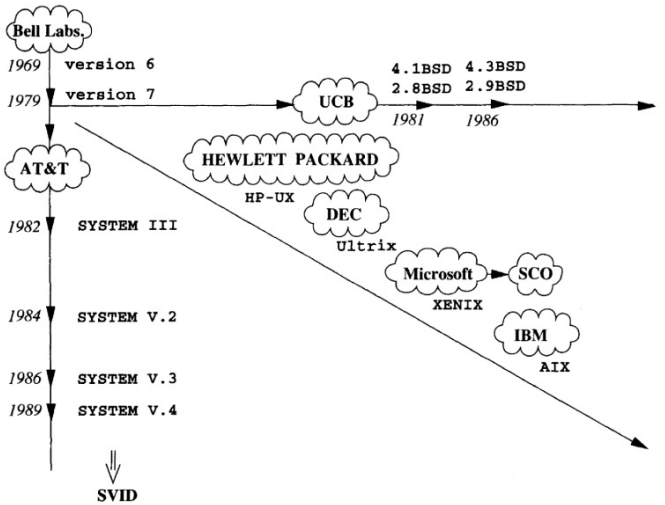
- Vznik operačního systému UNIX je datován r. 1969 a situován do střediska výzkumu firmy AT&T Bell Laboratories. K. Thompson, D. Ritchie. a B. Kernighan, pracující na projektu nového univerzálního operačního systému MULTICS, se snažili vytvořit příjemnější programovací prostředí, než právě MULTICS nabízel. Skupina odpadlíků poměrně rychle a z vlastní iniciativy vytvořila zárodek dnešního UNIXu na PDP-7. Vedení Bell Labs ale odmítlo dále tento projekt podporovat a teprve po návrhu vývoje prostředí pro práci s dokumentací v UNIXu pro patentové oddělení, vývoj financovala a podpořila počítačem typu PDP-1 1 .

- V průběhu několika let Ritchie vyvinul beztypový programovací jazyk BPCL, ze kterého po zavedení datových struktur vznikl jazyk C. Vývoj jazyka C byl veden výhradně záměrem pro přepis UNIXu do vyššího programovacího jazyka a motivován snahou o přenositelnost. V této době se UNIX začíná skutečně používat v patentovém oddělení, ale i dalších odděleních Bell Laboratories. Koncem 70. let se zdá být základní verze operačního systému podle představ autorů hotova a Bell Labs. po dohodě se státní správou a školstvím uvolňují UNIX version 7 pro použití i mimo výzkumné středisko. UNIX version 7 (označovaná zkráceně v7) má z pohledu uživatele všechny charakteristické rysy dnešních verzí. Je to zejména:
  - hierarchická struktura adresářů,
  - práce s procesy jako s elementy nositele změn v datové základně,
  - podpora volání jádra pro odstínění uživatele od konkrétního technického zařízení,
  - textově orientovaná interaktivní práce více uživatelů u terminálů.

- Základním testem pro UNIX byl přenos v7 na počítač jiného typu, a to INTERDATA 8/32, který potvrdil přenositelnost podle proklamovaných tezí. Následující vývoj probíhá pod vedením jednak AT&T a jednak University of California v Berkeley (UCB).
- AT8T vyvíjí snahu dopracovat UNIX do podoby komerčně dodávaných systémů pro profesionální použití, UCB se věnuje vývoji další systémové nadstavby, především v oblasti počítačových sítí.
- UNIX SYSTEM V je dominující verze, která je odrazovým můstkem současným implementacím na různých počítačích od různých výrobců. V průběhu 80. let se objevuje velké množství operačních systémů, které vzhledem k drahé licenci na zdrojové texty se chovají jako UNIX, ale není zaručena přenositelnost v nich odladěných programů do jiných systémů typu UNIX. AT&T proto vydává doporučení SVID (System V Interface Definition), které stanovuje základní přenositelnost programů pro UNIX na úrovni zdrojových textů vzhledem k voláním jádra, jež normalizuje. Dnes známá SVID3 je obecně uznávaným základním dokumentem pro UNIX a zabývá se specifikací i na úrovni příkazových řádků nástrojů programátora, konvencí jazyka C, atd.

- Distribuce systémů s označením BSD (Berkeley System Distribution) z UCB byla zahájena na počítačích VAX 5 označením 4.xBSD, kde x je označení verze a 2.xBSD pro počítače PDP všech modelů. Tým odborníků s označením BSD je dodnes světově respektovanou vývojovou skupinou ovlivňující nové směry UNIXu.
- Z obou větví čerpající světoví výrobci počítačů jsou na obrázku uvedeni v obláčcích, pod kterými je označení jejich komerčně dodávaného operačního systému UNIX, vždy vyhovující doporučení SVID. Přestože obyčejný uživatel rozpozná jinou implementaci pouze podle jména, jsou systémy pojmenovány různě, a to i přes úsilí sjednotit všechny tyto systémy pod jménem UNIX.
- V normalizačních snahách hraje důležitou roli evropská skupina výrobců X/OPEN (zal. r 1984), vydávající doporučení (X/OPEN Portability Guide), které se v podstatných partiích zcela shoduje s SVID.
- Normu POSIX vlastní lEEE (Institute of Electrical Electronic gineers v U.S.A.) pod evidencí Standard 1003. 1 — 1988 a definuje standardní rozhraní operačního systému a prostředí na bázi UNIX. POSIX je ale obecnější normou, snažící se definovat přenositelnost programů na úrovni zdrojových textů tak, aby byly přenositelné mezi různými typy operačních systémů.

Vývojové větve systému UNIX



Použití Unixu

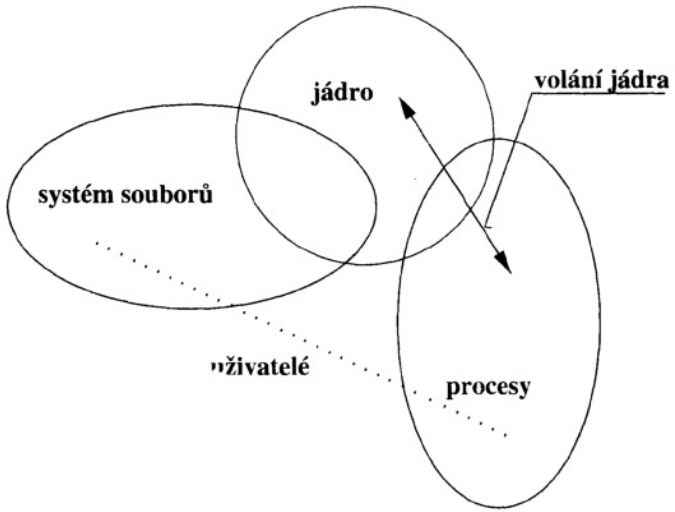
- Uživatelé komunikují s operačním systémem pomocí příkazového řádku. Budeme-li chtít zobrazit obsah souboru se jménem např. `.profile`, napíšeme:

```
$ cat .profile
```

- V uvedeném příkladu příkazového řádku pro zajištění našeho požadavku vytváří UNIX proces, který provádí příkaz `cat` a manipuluje se souborem `.profile`
- Oba dva fenomény, soubor i proces, jsou podporovány ve své existenci fenoménem třetím, kterým je **jádro** (angl. **kernel**).
- Jádro je program, který je zaveden po zapnutí počítače do operační paměti a tam je spuštěn. Je programem, který z počítače na úrovni technického vybavení vytváří virtuální počítač, odstíní uživatele od přímého styku s technickým vybavením, umožňuje se strojem komunikaci ve formě vyššího programovacího jazyka a dokáže zajistit sdílení technických zdrojů několika uživatelům najednou.

- Pro uchování dat na vnějších paměťových médiích (zejména magnetických discích) zajišťuje přístup ke struktuře dat nazývané systém souborů (angl. filesystem). Pro uživatele to znamená, že může svá data ukládat do souborů a opět je ze souborů vybírat pro další zpracování. Proces je realizace akce uživatele nebo systému. Je nositelem změn v datové základně konkrétní instalace operačního systému.
- Procesem je např. provedení kopie souboru na tiskárnu, překlad z úrovně zdrojového textu programu do úrovně strojového kódu, spuštění databáze atd.
- Jsou to volání jádra (System Calls), pomocí kterých procesy interagují s jádrem. Volání jádra má z hlediska psaní programu tvar volání knihovní funkce; je to např. žádost o zpřístupnění dat z disku, žádost o vytvoření nového procesu atd.
- Schématicky lze základní tři komponenty systému zobrazit :

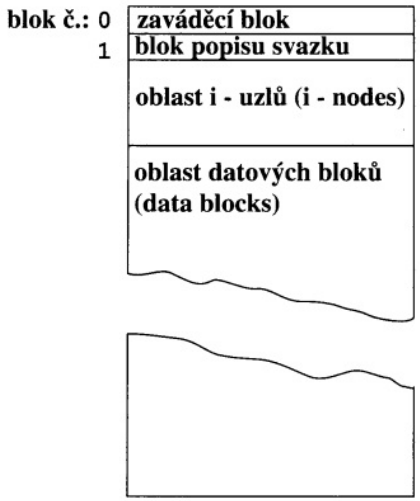
Jádro, procesy, systém souborů



Systém souborů

- Systém souborů je realizován na magnetickém médiu. Logická struktura magnetického média je svazek (angl. je používán výraz filesystem, v překladu systém souborů). Logiku vnitřní struktury svazku ukazuje obrázek.
- Z obrázku je výraz i-uzel. I-uzel je jednoznačná identifikace souboru, obsahuje jeho atributy, a že s číslem i-uzlu je v adresáři spojeno jméno souboru. Číslo i-uzlu je jeho pořadí v rámci oblasti i-uzlů. Velikost i-uzlu je 64 slabik a i-uzel obsahuje tyto informace:
  - vlastník souboru,
  - typ souboru (obyčejný soubor, adresář ...)
  - přístupová práva,
  - datum a čas poslední manipulace se souborem,
  - počet odkazů na soubor z různých míst stromu adresářů,
  - tabulka datových bloků,
  - velikost souboru.

Struktura svazku

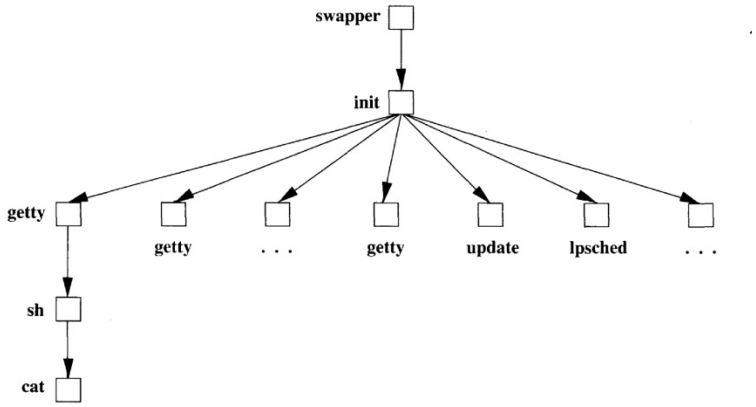


Procesy

- Obdobně jako systém souborů, i procesy tvoří hierarchickou strukturu. Je-li nastartován operační systém, tj. je-li spuštěno jádro, vzniká za jeho podpory proces nazývaný `swapper` (někdy `sched`). Je to proces, který je identifikován jako proces č. 0. Jeho hlavním smyslem je pracovat pro údržbu správy paměti, ale také, ihned na začátku své činnosti, sám vytváří další proces pomocí odkazu na jádro.
- Tento další proces je nazýván `init` a má identifikační číslo 1 . Proces, který vznikne odkazem na jádro z jiného procesu, nazýváme procesem synovským (**child**) vzhledem k procesu, který jej takto vytváří a který je označován jako rodič (**parent**).
- Žádosti o vznik dalšího procesu říkáme **`fork`** (rozvětvení) a proces jej uplatní pomocí volání jádra `fork(2)`.
- Proces `init` je otcem dalších procesů. `Init` udržuje několik úrovní stavu systému, přitom je každá úroveň reprezentována množinou procesů, které `init` vytváří a dohlíží na jejich stav.

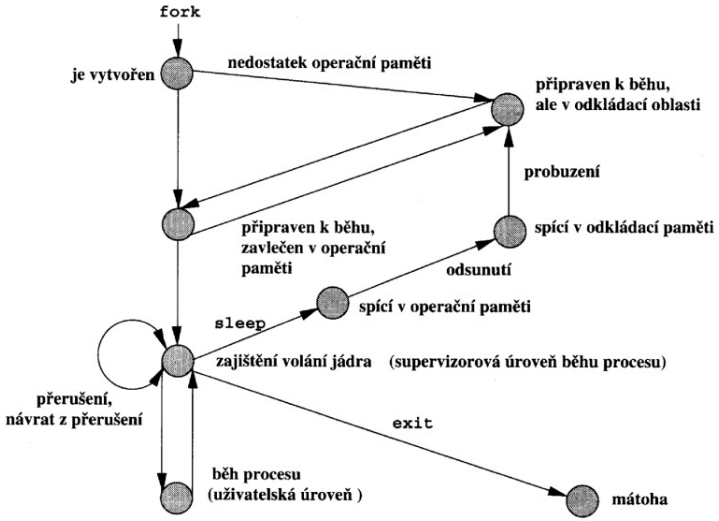
- Hlavní dvě úrovně jsou označovány jako úroveň jednouživatelská (**single user**) a úroveň víceuživatelská (**multiuser**). Na úrovni jednouživatelské vytváří `init` jen několik dalších procesů, které umožňují vstup do systému pouze jednomu uživateli, a to privilegovanému. Je to úroveň pro údržbu a celkové změny v systému.
- Úroveň víceuživatelská je standardní úroveň pro běh systému, kdy se interaktivně z terminálů uživatelé přihlašují do systému a pracují v něm.
- Aby proces `init` umožnil uživateli vstup do systému, vytváří proces, jehož standardním vstupem i výstupem je terminál. Takový proces je nazýván `getty` a komunikuje s uživatelem v době přihlašování. Procesům `getty` (jejich počet je dán počtem terminálů) jádro přiděluje pro jednoznačnou identifikaci poslední přidělené číslo procesu zvýšené o 1 . Proces `getty` se v případě správného přihlášení mění na proces `sh`, který realizuje běžnou komunikaci uživatele se systémem a je nazýván příkazovým interpretem (např. Bourne shell), `sh` na obrazovce vypisuje znak ' $ ' (nebo ' # ') a interpretuje příkazový řádek uživatele.

Hierarchická struktura procesů



- Dříve než může být procesu přidělen procesor, musí být proces zaveden do operační paměti. Ve chvíli, kdy je proces vytvořen, je snahou jádra přidělit mu požadovanou paměť.
- Není-li dostatečně velká požadovaná paměť volná, jádro uplatňuje na procesy v operační paměti algoritmy se snahou odsunout některé procesy mimo paměť a zajistit tak dostatek volné paměti pro nový proces. Orientace jádra zde vychází z posouzení doby procesu již strávené v paměti.
- Proces nejdéle sídlící v paměti je odsunut. Místo, kam jsou procesy z paměti odsouvány (swap out) a odkud jsou opět po čase do paměti zavlékány (swap in), je disková paměť označovaná jako odkládací oblast (swap area).
- Odkládací oblast je dána parametry jádra a je situována mimo jakýkoliv svazek systému souborů. Pro práci s odkládací oblastí jádro používá proces s PID=0 označovaný jako `swapper`.
- Stavy, ve kterých se může proces od svého vzniku do svého ukončení nacházet, lze zobrazit následujícím diagramem:

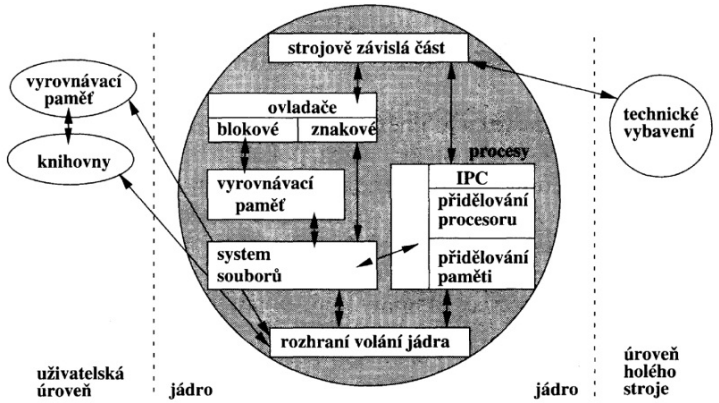
Stavy procesu



Jádro

- Jádro (kernel) je programové vybavení, které pracuje na počítači bez jakékoliv další programové podpory. Je velkým rozhraním mezi uživatelem a technickým vybavením výpočetního systému. Zajišťuje realizaci odkazů uživatelů nebo procesů na periferie a vůbec všechny technické zdroje počítače. Dále udržuje a podporuje v činnosti systém souborů a běžící procesy. Celkovou strukturu jádra ukazuje obrázek.
- Uživatelská úroveň je jádrem podporována přes rozhraní volání jádra. Stručně řečeno, program používá volání funkce, která je voláním jádra, a program tak vstupuje do režimu obsluhy jádrem (proces vstupuje do supervizorové úrovně). Činnost jádra v okamžiku, kdy zajišťuje určitou akci vyžádanou procesem, je jiným procesem nepřerušitelná. Vzhledem k časové prodlevě, po kterou proces v supervizorové úrovni zůstává, není vhodný pro řízení procesů v reálném čase.
- Pro řízení v reálném čase se používají speciálně navržené unixové systémy

Struktura jádra



Procesy

- Proces je základním elementem chodu systému. Uživateli je k dispozici volání jádra `fork(2)` , které má formát:

```
int fork(void);
```

- Jádro vytvoří nový proces, který je
  - identický s procesem, který jej vytváří, tzn.: je vytvořen nový proces, jehož instrukční segment, datový segment i zásobník je týž jako segmenty volajícího procesu, pouze je odlišen novým PID,
  - provádění nově vytvořeného procesu pokračuje od místa volání `fork(2)`,
  - nově vzniklý proces je označován jako syn volajícího procesu, který je otcem.
- Od této chvíle otec i syn pracují nad týmž instrukčním kódem. Z návratové hodnoty `fork(2)` může proces identifikovat svoji totožnost. Je-li návratová hodnota 0 , proces je syn, tedy nově vzniklý proces, je-li návratová hodnota nenulové kladné číslo, proces je otec a návratová hodnota odpovídá PID syna.

```
extern int errno;   /* V případě kolize je možné identifikovat důvod */
perror(retez)    /* funkce panic error */
char *retez;
{
    fprintf(stderr, "%s\n" , retez);
    exit (-1);
}
main()
{
    switch(fork() ) {
     case -1 : perror("Chyba);
           break;
     case 0 : printf("Syn\n");
           break;
     default : printf("Otec\n");
           break;
     }
    exit(0);
}
```

Synchronizovat lze procesy pomocí volání jádra `wait(2)`. V předchozím příkladu nelze dopředu jednoznačně říct, který proces svoji identifikaci vypíše jako první (záleží to na modulech jádra pro správu přidělování paměti). Otec ale může na svého syna čekat. V tomto případě můžeme v uvedeném příkladu ve větvi `default` psát

```
....
main()
{
    int status;
    switch( fork() ) {
      ……
     default : wait(&status);
     printf ( "Otec\n" );
     break;
    ……..
```

Volání jádra `execl(2)` umí přepsat textový a datový segment volaného procesu textovým a datovým segmentem, který odpovídá programu v argumentech volání. Např.

```
main()
{
int chpid, status;
switch ( (chpid=fork() ) ) {
    case -1: perror("Chyba!");
    case 0: execl("/bin/ls', "ls", "-l", NULL);
    default: while(wait(&status) != chpid);
     break;
    }
……
}   /* konec main() */
```

otec čeká na ukončení svého syna, který provede výpis obsahu pracovního adresáře.

Formát volání exec (2) má několik tvarů např:

```
int execl(const char *path const char *arg0, const char
*argl, .., const char *argn, (char *)0);
```

kde argument `path` je cesta souboru s programem, jehož textový segment je nahrazen stávajícím, argumenty `arg0, arg1, ..., argn` jsou parametry odpovídající řetězcům pole `argy` funkce main ( ) ve spuštěném procesu. Výčet parametrů je zakončen `NULL` specifikací.

Signály

- Upozorňovat se navzájem na vznik určité situace mohou procesy pomocí signálů, a to pomocí volání jádra `kill(2)` . Má formát:

```
int kill(int pid, int sig);
```

- přitom signál `sig` je zaslán procesu s identifikačním číslem pid (PID).
- V příkladu PID syna využívá otec k zaslání signál č. 2 po 10 vteřinách čekání. Funkce `sleep(3)` je standardní, ve svém argumentu obsahuje počet vteřin, po který je proces pozastaven. Signál č. 2 odpovídá pokynu přerušení z klávesnice, proces syn je tehdy zrušen.
- Signál může být také zaslán synem otci. Syn zjišťuje PID svého otce voláním jádra `getppid(2)` . Proces může své vlastní PID zjistit voláním jádra `getpid(2)`

```
int getpid(void);
int getppid(void);
```

```
main()
{
int pid;
    switch(pid=fork() ) {
     case -1 :  /* chyba při vytváření syna */
          exit(-1);
     case 0 : /* syn aktivně čeká ve smyčce */
          for ( ; ; );
     default : /* otec po 10 sekundách */
          sleep(10);
          /* posílá synu signál č. 2 */
          switch( kill(pid, 2)) {
               case -1 : perror("Chyba kill() !");
               case 0 : printf("O.K.\n");
                    break;
          }
 }
 exit(0);
}
```

Předchozí příklad můžeme obměnit tak, aby signál posílal syn otci takto, kde otci syn posílá signál č. 9 (nejsilnější výzva k ukončení).

```
main()
{
   switch(fork()) {
    case -1: perror("Chyba fork() !");
    case 0: sleep(10);
       switch(kill(getppid() , 9)) {
          case -1 : perror("Chyba kill() !');
          /* je nutno otce zastavit jinými prostředky
          */
          case 0 : printf("O.K.\n");
             break;
       }
       break;
    default : for ( ; ; ) ;
   exit(0);
}
```

Tabulka signálů

číslo	označení	implicitně	význam
1	SIGHUP	ukončení	odpojení terminálu,
2	SIGINT	ukončení	přerušení z klávesnice,
3	SIGQUIT	*	konec s uložením obrazu paměti,
4	SIGILL	*	neznámá instrukce,
5	SIGTRAP	*	ladící přerušení,
6	SIGABRT	*	ukončení z důvodu v/v,
7	SIGEMT	*	instrukce EMT,
8	SIGFPE	*	kolize reálného čísla,
9	SIGKILL	ukončení	okamžité ukončení procesu,
10	SIGBUS	*	kolize sběrnice,
11	SIGSEGV	*	selhání segmentace,
12	SIGSYS	*	chybný tvar volání jádra,
13	SIGPIPE	ukončení	zapisovanou rouru nikdo nečte,
14	SIGALRM	ukončení	konec časového intervalu,
15	SIGTERM	ukončení	ukončení,
16	SIGUSR1	ukončení	první signál definovaný uživatelem,
17	SIGUSR2	ukončení	druhý signál definovaný uživatelem,
18	SIGCHLD	ignorování	změna stavu synovského procesu
19	SIGPWR	ignorování	kolize zdroje,
20	SIGWINCH	ignorování	změna velikosti okna,
21	SIGPOLL	ukončení	příznak při práci s PROUDY,
22	SIGSTOP	pozastavení	signál pozastavení procesu,
23	SIGTSTP	pozastavení	pozastavení procesu uživatelem,
24	SIGCONT	ignorování	pokračování v činnosti,
25	SIGTTIN	pozastavení	čekání na vstup,
26	SIGTTOU	pozastavení	čekání na výstup,
27	SIGXCPU	*	konec časového kvanta procesoru,
28	SIGXFSZ	*	překročení stanovené délky souboru.

Komunikace mezi procesy

- Procesy mohou komunikovat na základní úrovni pomocí signálů. Tento způsob je bohužel omezen pouze na upozornění na určitou událost a reakci na ni. Dále se u procesů předpokládá znalost svých PID.
- Dva procesy mohou spolu komunikovat na úrovni předávání dat pomocí **roury** (**pipe**). Roura je paměťová oblast sdílená dvěma procesy tak, že jeden proces do roury pouze zapisuje a druhý proces z ní pouze čte. Jedná se o frontu FIFO (First In Eirst Qut), to znamená, že čtoucí proces obdrží nejprve data, která zapisující proces poslal do roury jako první.
- V mnemonice Bourne shell můžeme schématicky psát

  `PRODUCENT | KONZUMENT`

  kde znak '|' je symbol roury a proces `PRODUCENT` do roury zapisuje, kdežto `KONZUMENT` z roury data čte.
- Proces může vytvořit rouru voláním jádra

  ```
  int pipe(int pd[2]);
  ```

  a získává tím deskriptor čtení z roury pd[0] a deskriptor zápisu pd[1].

- Realizaci příkladu příkazového řádku Bourne shell

  `ls | wc –l`

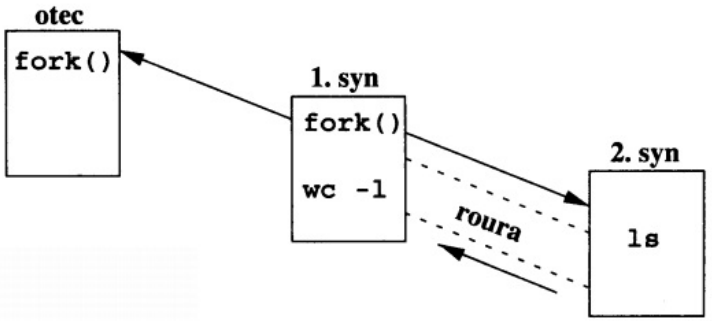
  jehož výsledný efekt je výpis počtu položek pracovního adresáře na standardní výstup. Napíšeme text programu, kde vzniknou dva synovské procesy a každý z nich bude realizovat jeden z programů `ls` a `wc`.
- Spuštěný proces vytváří syna (je označen jako 1 . syn) a čeká na jeho dokončení. Po dokončení vrací tentýž návratový status. Syn vytvoří rouru voláním jádra `pipe(2)` a voláním `fork(2)` svého syna (2. syn). Chová se dále jako KONZUMENT, uzavírá svůj standardní vstup a voláním `dup(2)` si jako svůj standardní vstup připojí deskriptor roury pro čtení, uzavře rouru pro zápis a čtením svého standardního vstupu bude nyní připraven odebírat data z roury. Nakonec sám sebe přepíše programem `wc(1)` . Proces pokračuje dál podle textu `wc(1)` , zdědil přitom atributy svého otce; vstup programu `wc(1)` bude tedy přesměrován na vstup roury.

- Jeho syn (označován jako 2. syn) se naopak chová jako PRODUCENT, zdědil kromě všech otevřených kanálů také deskriptory vytvořené roury, uzavírá svůj standardní výstup, voláním jádra `dup(2)` si jako svůj standardní výstup připojí deskriptor roury pro zápis, uzavře rouru pro čtení, a pokud nyní bude zapisovat na svůj standardní výstup, zapisuje do roury. Přepisuje se poté programem `ls(1)`, který dědí všechny jeho atributy. Tím jsou oba komunikující procesy vytvořeny a současně je mezi nimi vytvořen jednosměrný komunikační datový kanál zvaný **roura**. Oba komunikující procesy v příkladu uzavírají nepoužitý konec roury, aby se rozpoznalo, kdy PRODUCENT končí. Jádro rozpozná konec činnosti PRODUCENTa teprve tehdy, až zápisový konec roury není otevřen pro žádný proces. PRODUCENT je vytvářen jako druhý syn, protože ukončení jeho činnosti, které bude následovat za uzavřením roury, znamená ukončení činnosti všech jeho synů; v opačném případě by to znamenalo i ukončení činnosti `wc(1)` .

```
#include <stdio.h>
int pd[2];
int status;
main() {
    switch(fork()) {
     case -1: exit(1); /* Chyba */
     case 0: /* první syn - konzument, realizuje program wc */
          pipe (pd);
          switch(fork()){ /* první syn vytváří svého partnera pro
          komunikaci */
               case -1 : exit (-1) ; /* Chyba 1 . syna */
               case 0 : /* druhy syn – producent, realizuje program ls  */
                    close(1);
                    dup(pd[1]);
                    close(pd[0]);
                    execl ( " /bin/ls" , „ls" , NULL );
```

```
            default : /* první syn */
                close(0);
                dup(pd[0]);
                close(pd[1]);
                execl („/bin/wc“ , “wc“ , “ -l “ , NULL);
            } /* konec switch() při vytvoření druhého syna */
        default : /* Otec čeká na ukončení komunikace synů /
            wait (&status);
            exit(status & 0377);
        } /* konec switch() */
    } /* konec main() */
```

Roura “ls | wc -l”



Roura dvou synů stejné úrovně