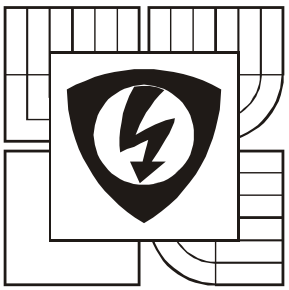
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



Síťové operační systémy

Garant předmětu:
Dan Komosný

Autoři textu:
Dan Komosný a kolektiv

BRNO 2022

Obsah

1 INFORMACE O PŘEDMĚTU

Cílem předmětu je poskytnout studentům obecný pohled na problematiku síťových operačních systémů. Základní publikace, které byly použity pro sestavení tohoto kurzu, jsou Network Operating Systems [16], Modern Operating Systems [11] a Understanding Operating Systems [30]. Obsahově předmět spadá do **informatické** části vyučovaných oborů. Účelem předmětu je studenty seznámit s **principem činnosti operačních systémů**. Tyto znalosti lze dále využít ve velkém množství navazujících oblastí, které zahrnují operační systémy. Jedná se například o správu sítí, operačních systémů síťových prvků, operačních systémů koncových stanic, programování komunikace mezi procesy/vlákny, návrh a úprava operačních systémů pro komunikační jednotky senzorových sítí, řešení komunikace mezi operačními systémy reálného času v leteckých systémech, správa operačních systémů VoIP ústředen atd. Účelem předmětu je, aby získané znalosti o principu činnosti operačních systémů bylo možno dále rozvinout pro konkrétní oblast využití.

Předmět je dělen do tří tematických částí týkajících se síťových operačních systémů:

- obecné základy,
- síťová část,
- služby a bezpečnost.

První a nejrozsáhlejší část je zaměřena na teorii týkající se činnosti operačních systémů obecně. Je zde probírána architektura operačních systémů, koncept procesů, správa paměti a souborové systémy. V druhé části následuje popis síťové části operačních systémů, včetně vysvětlení funkce démonů a superdémona. Poslední část předmětu se zabývá základními službami a bezpečností operačních systémů. Probrány jsou mimo jiné základní typy útoků a princip firewallu/proxy.

Základní teorie stavby a činnosti operačních systémů je podána **obecně**. Pro snazší pochopení určitých částí je popis teorie podpořen **příklady**, především z operačního systému Linux. Jedná se o výpisy z příkazového řádku[1]. S popisem teorie se zastoupení příkladů ve skriptech postupně navyšuje.

V rámci praktických cvičení jsou studenti seznámeni s uživatelským rozhraním operačního systému GNU/Linux, od základních příkazů ke složitějším operacím údržby a konfigurace systému. Jedná se o operace jako jsou např. evidence uživatelů, nastavení uživatelských práv, konfigurace síťových prostředků, využití nástrojů správy sítě atd. Praktická cvičení jsou koncipována tak, aby se studenti nejdříve seznámili s potřebnými nástroji a příkazy. V průběhu semestru jsou pak postupně počítačová cvičení složitější v tom, že je nutno již provádět některé úkony samostatně. Každé počítačové cvičení je ukončeno samostatnou úlohou, kde si studenti ověří a upevní nabyté znalosti z daného cvičení.

Na těchto skriptech se podíleli i další autoři, zejména Michal Soumar, Patrik Morávek a Jiří Balej. K úrovni a obsahu skript také přispěli studenti předmětu na základě jejich připomínek a námětů. Skripta jsou průběžně aktualizována a proto vždy pracujte s **aktuální verzí**.

[1] Ve skriptech jsou použity i příklady zapsané pomocí syntaxe jazyka C/C++. Tyto příklady byly pro potřeby skript upraveny do formy částečného pseudokódu.

2 ÚVOD

2.1 Základní koncept

Možnost síťové komunikace se stala standardem a doba, kdy se používaly izolované operační systémy, je minulostí. Síťové operační systémy[1] zpravidla pracují na serverech, které poskytují služby pro uživatele připojené pomocí sítě. Mezi základní služby patří sdílení souborových systémů a databází, podpora emailu a webových stránek. Operační systémy na klientských stanicích zajišťují přístup k těmto službám.

Začněme první věcí, která by měla být definována před začátkem studia tohoto textu. Jedná se o to, co operační systém (OS) je a k čemu se používá. Můžeme říci, že operační systém je zprostředkovatel mezi uživatelem a výpočetním prostředkem. Z pohledu uživatele je účelem operačního systému poskytnout prostředí, ve kterém je možno spouštět programy. Uživatel těmto programům předává vstupy a jako výsledek běhu programu očekává výstupy. S operačním systémem lze pracovat **efektivně** – s nějakou bližší znalostí struktury a principů operačního systému, nebo **snadně** – bez velkých nároků na znalost operačního systému. U různých operačních systémů některý z těchto dvou aspektů převládá, buď je preferována efektivita nebo snadnost.

Příkladem může být použití grafického rozhraní (GUI – Graphical User Interface). Některé operační systémy poskytují grafické rozhraní, které zjednodušuje práci. Lze tak snadně spouštět programy, konfigurovat operační systém, služby a programy. Grafické rozhraní může být pevně zabudované, nebo může být provozováno jako volitelná součást. Na druhou stranu platí, že pro profesionální práci při správě serverů a síťových zařízení[2] (směrovače, přepínače) je preferováno textové rozhraní (příkazový řádek). Důvody jsou prosté – textové rozhraní umožňuje provádět konfigurace „šité na míru“. Umožňuje také snadno automatizovat postupy pomocí skriptů, např. zavedení velkého počtu uživatelských účtů do systému, instalace programů na vetší počet stanic. U serverů není také typicky grafické rozhraní provozováno, protože by zbytečně zabíralo systémové prostředky. Zde tedy převládá efektivita nad snadností (uživatelskou přívětivostí). Za zmínku stojí, že u některých operačních systémů nelze použít plnohodnotné grafické rozhraní z důvodu omezených prostředků. Pro tyto systémy existují zjednodušená grafická rozhraní, která zabírají málo prostředků. Další příkladem snadnosti vs. efektivity práce při konfiguraci operačního systému jsou senzorové jednotky, kde je kladen důraz na omezený hardware z důvodu nízké ceny zařízení a případného napájení z baterie (energetická náročnost), tj. snadnost práce zde není prioritou.

[1] Pojem *síťový operační systém* není napříč používanou literaturou využitou při psaní těchto skript jednoznačně definován – různé zdroje uvádí jiné definice. Převládá však přístup, že síťový operační systém je takový systém, který pracuje na serveru a poskytuje služby (nesíťovým) operačním systémům, které pracují na klientských stanicích. Ovšem za síťový operační systém lze považovat i takový systém, který pracuje na mezilehlých síťových prvcích zajišťujících komunikaci v síti (například směrovač a přepínač). My se v textu budeme primárně držet definice síťového operačního systému jako systému pracujícího na serveru, který poskytuje služby klientům.

[2] Výkonná zařízení, ne domácí použití.

Na otázku, co vlastně je operační systém z pohledu systému, není snadné odpovědět. Důvodem je fakt, že operační systém vykonává dvě základní funkce, které jsou od sebe odlišné [11].

První funkce operačního systému je pracovat jako **rozšířený automat** (extended machine). Tento automat umožňuje jednodušší obsluhu výpočetního systému než složitá práce se samotným hardware. Toto je zařízeno tak, že operační systém poskytuje množinu služeb, ke kterým mají programy přístup pomocí **systémových volání**. Programátora tak nezajímá, jak je řešen přístup ke konkrétnímu hardware (například paměťovému médiu), ale je mu poskytnuto jednoduché rozhraní na vyšší úrovni (z pohledu nejnižší úrovně, kterou je hardware). Výsledkem je, že jsou uživateli poskytovány funkce pro práci s hardware[3]. Jedná se například o otevření, čtení, zápis a uzavření souboru uloženého na paměťovém médiu. Detailů, například ohledně konkrétní pozice dat na paměťovém médiu, je pak uživatel ušetřen. Dalším příkladem tohoto pohledu na operační systém může být skrytí detailů, jako je například obsluha přerušení, správa paměti atd. Abstrakce používá přístup „ze shora dolů“ (top-down view).

Druhá funkce operačního systému je pracovat jako **správce zdrojů** (resource manager). Tato funkce vychází z opačného přístupu, než jaký byl popsán předešle. Správa zdrojů vychází z přístupu od nižších vrstev směrem nahoru (bottom-up view). Počítačové systémy zahrnují mnoho částí/zdrojů, jako je například procesor (CPU), paměť, časovače, paměťová média, síťová rozhraní atd. Těmito zdroji nemusí být jen hardware, ale také informace (soubory, databáze atd.). Úkolem operačního systému je poskytnout programům přístup k těmto zdrojům. Můžeme tedy říci, že operační systém udržuje informaci o tom, který proces[4] využívá který zdroj. Dále operační systém umožňuje změnu procesů při využívání zdrojů. Z toho vyplývá i další požadavek na operační systém – řeší konflikty při (pokusu) využívání jednoho zdroje současně více procesy. Využívání zdrojů je řešeno ve dvou rovinách – **časová** a **prostorová**.

V časové rovině se procesy o daný zdroj postupně střídají. Příkladem může být střídání o čas procesoru. Tím je umožněno, že ve stejném čase může být spuštěno více procesů, které se v krátkých časech střídají o procesor (víceúlohový operační systém). Dalším příkladem střídání o jeden zdroj je postupný tisk dokumentů na tiskárně. V prostorové rovině procesy používají část daného zdroje. Příkladem je hlavní paměť (RAM), která je rozdělena dílčí bloky. V těchto blocích jsou uložena data procesů (celý proces nebo jeho část). V paměti je tak uloženo více procesů současně.

Shrňme si tedy funkce, které poskytuje operační systém:

- Spouštění programů – uživatel předává vstupy a očekává výstupy.
- Abstrakce hardware – jednotné rozhraní pro programy při práci s hardware (např. API[5]).
- Správa zdrojů – přiděluje/odebírá/řeší konflikty při využívání zdrojů procesy.

Umístění operačního systému z pohledu návaznosti dalších částí výpočetního prostředku je zobrazeno na obrázku 2.1 [11]. Ve spodní části se nachází hardware, který je

[3] Systémová volání umožňují i jiné funkce, které nejsou svázány s hardware.
[4] Proces je instance spuštěného programu. Více o procesech si uvedeme dále.
[5] Application Programming Interface.

tvořen několika vrstvami. Spodní vrstva zahrnuje vlastní fyzická zařízení. Vrstva **mikroarchitektury** seskupuje fyzické prvky do skupin, které poskytují určité funkce. Typicky jsou to registry procesoru, sčítače, násobiče a čítače. Mikroarchitektura implementuje **instrukční sadu** ISA (Instruction Set Architecture). Jedná se o sadu pro ovládání hardware, která typicky zahrnuje instrukce procesoru, registry, datové typy atd. Stejné **architektury systémů** používají stejnou instrukční sadu. Příklady architektur mohou být IA-32 (také označováno jako i386), x86-64 a ARM. Stejná architektura může být realizovaná různými typy procesorů, které se však odlišují provedením mikroarchitektury. Program přeložený do instrukční sady dané architektury lze tedy spustit na různých procesorech. Instrukční sada typicky obsahuje 50 až 300 instrukcí, například pro přesun dat, provádění aritmetických operací a porovnání hodnot. Formát zápisu je číselný kód instrukce následovaný operandy, které mohou být kód registru, adresa paměti a hodnota. Posloupnost instrukcí a operandů je označován jako **strojový kód**. Pro přímé programování architektury se používá jazyk **symbolických instrukcí a adres** (assembly language). Tento jazyk používá symboly pro reprezentaci číselných kódů instrukcí a registrů. Symboly pak podle svého názvu indikují, co která instrukce provádí (např. přesun dat), či které místo v paměti se adresuje (název registru). Tyto symboly jsou specifické pro danou architekturu a jejich příklad je uveden ve výpisu 2.2, který provádí jednoduché porovnáni a odečet hodnoty. Překlad symbolických instrukcí na instrukční sadu architektury provádí program, který nazýváme **asembler**. Tento program ze symbolických instrukcí vygeneruje strojový kód, který má podobu sekvence číselných hodnot zobrazovaných v hexadecimální nebo binární podobě.

```
1       mov ax,0x30   ;uloz do ax hodnotu 0x30
2                     ;jako vysledek predchozi operace
3       cmp ax,0x25   ;porovnej ax s hodnotou 0x25
4       jl  END       ;jestlize mensi tak jdi na konec
5       sub ax,0x25   ;odecti hodnotu x25
6   END: mov ax,mx    ;vysledek do mx
```

Výpis kódu 2.1: Příklad jazyka symbolických instrukcí pro jednoduchou operaci.

Jedna ze základních činností sytému je práce s daty uloženými na paměťovém médiu. Čtení z paměťového média je zařízeno tak, že do registrů je uložen příkaz k provedení (čtení/zápis/dotaz na stav) a jeho parametry. Tyto parametry v podobě symbolů a hodnot pro čtení dat z plotnového disku jsou počet sektorů k načtení, cylindr ze kterého číst, počáteční sektor, hlava plotny, identifikátor disku a kam uložit data. Následně se vyžádá funkce programu BIOS (Basic Input-Output System), který realizuje základní vstupně/-výstupní rozhraní (označován také jako firmware). Program BIOS může být uložen na základní desce stroje s procesorem pevně bez možnosti změny, nebo v přepisovatelné paměti, která umožňuje jeho aktualizaci. Funkce programu BIOS jsou dostupné pomocí **přerušení**. Pro čtení dat z disku je vyvoláno přerušení číslo 0x13, které BIOS obslouží pomocí připravených dat v registrech, vrátí požadovaná data a také vrátí návratovou hodnotu vyvolaného přerušení. Návratová hodnota, která je uložena do registru, může značit úspěch, chybný příkaz, sektor nenalezen, detekován špatný sektor atd. Uvedený postup je ve zjednodušené podobě uveden ve výpisu 2.2.

```
1 mov ah,0x02    ;0x02 = funkce - chceme cist z disku
```

```
mov al,0x2     ;kolik sektoru nacist
mov ch,0x0     ;cylindr, ze ktereho cist
mov cl,0x0     ;prvni sektor ktery cist
mov dh,0x0     ;hlava disku
mov dl,0x80    ;prvni pevny disk
mov bx,0x5000 ;kam se maji data nacist (adresa pameti)
...
int 0x13       ;zavola preruseni c. 13 pro cteni dat
               ;navratova hodnota v registru ah (0 - uspech)
```

Výpis kódu 2.2: Jazyk symbolických instrukcí při čtení data z paměťového média.

BIOS umožňuje konfiguraci, provádí inicializaci a test hardware. Startuje operační systém tím, že z paměťového média načte **zavaděč**, který následně spustí systémové jádro[6]. Operační systém již pak pracuje s hardwarem přímo, jelikož je to rychlejší než využívání funkcí BIOSu. Nástupcem základního BIOSu je UEFI (Unified Extensible Firmware Interface), který například implementuje příkazové rozhraní (interpret příkazů, shell). To příkazové rozhraní lze mimo jiné použít pro jeho aktualizaci.

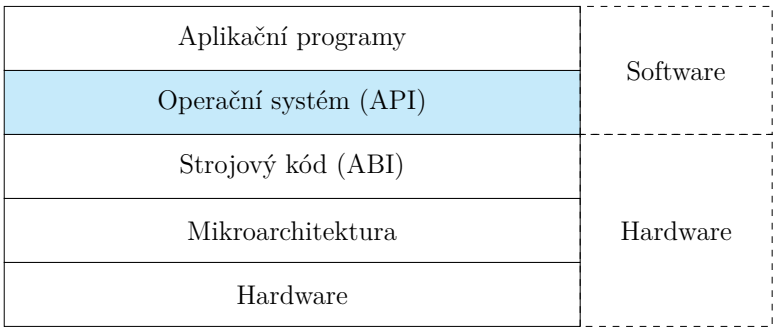
Program BIOS také umožňuje zobrazit informace o hardwarovém vybavení stroje, včetně sériových čísel. Výstup programu lze například číst pomocí příkazu `dmidecode` (dmi – Desktop Management Interface). Výpis 2.3 zobrazuje možnosti příkazu, kde první volbou jsou informace o vlastním programu BIOS. Následně jsou zobrazeny údaje o tomto programu, jako například velikost paměti ROM, ve které je umístěn, možnost jeho aktualizace a podpora UEFI. U vybraných zařízení jsou také zobrazena přerušení, pomocí kterých lze s nimi pracovat (podobně jako u výše uvedeného příkladu čtení dat z disku).

```
[]$ man dmidecode
Type  Information
0     BIOS
1     System
2     Baseboard
3     Chassis
4     Processor
...
[]# dmidecode --type bios
BIOS Information
 Vendor: American Megatrends Inc.
 ROM Size: 16 MB
 Characteristics:
  PCI is supported
  BIOS is upgradeable
  Boot from CD is supported
  Selectable boot is supported
  Print screen service is supported (int 5h)
  Serial services are supported (int 14h)
  Printer services are supported (int 17h)
  UEFI is supported
```

Výpis kódu 2.3: Zjištění informací o hardwarovém vybavení a programu BIOS.

[6] Jádro bude definováno následně.

Nad vrstvou strojového jazyka se nachází operační systém. Struktura přeložených programů (jejich binární reprezentace) v operačním systému je definována pomocí **rozhraní ABI** (Application Binary Interface). S tímto rozhraním pracují kompilátory zdrojových kódů. ABI popisuje jak se má přistupovat a pracovat s prostředky dané architektury systému. Například při volání operace definuje, kam mají být uloženy vstupní parametry a kde je poskytnuta návratová hodnota (registry, zásobník). ABI je využito při volání funkcí přeložených knihoven či jádra systému (viz dále), tím že definuje jejich rozhraní na úrovni strojového jazyka. Příkladem ABI může být rozhraní volací konvence x86 pro architektury IA-32 (i386) a x86-64. Naproti tomu **rozhraní API** (Application Programming Interface) poskytuje způsob pro využívání funkcí knihoven či jádra ve zdrojovém kódu a je využíváno programátorem. API je popisováno pomocí zvoleného programovacího jazyka (C++, Python) a je nezávislé na architektuře systému. Příkladem API je využívání funkcí operačního systému – POSIX a Windows API.



Obrázek 2.1: Umístění operačního systému.

2.2 Historie a typy

Stěžejní kroky při vývoji operačních systémů byly tyto:

- abstrakce systémových prostředků,
- interaktivní a dávkový režim,
- práce více uživatelů a řešení více úloh současně,
- přenositelnost mezi různými hardwarovými platformami,
- přiblížení běžným uživatelům.

Počítače první generace, které vznikaly krátce po 2. světové válce, operační systém neobsahovaly. Programy pro tyto počítače se psaly v jazyce symbolických instrukcí a obsluha musela vědět, na kterých vstupech a výstupech může operovat (registry). Program byl uložen na propojovacích deskách, později se používaly magnetické pásky. K usnadnění práce bylo následně vyvinuto několik **programovacích jazyků**[7]. Programy napsané v těchto jazycích bylo nutné zkompilovat (přeložit do strojového jazyka) na příslušný počítač. Stále bylo nutné adresovat vstupní/výstupní místa v paměti. Proto bylo vhodné vyvinout program, který by vnitřní funkce systému prováděl automaticky bez potřeby

[7]Například Fortran a COBOL.

jejich detailních znalostí. Začaly vznikat první operační systémy, které zprostředkovávaly hardware uživateli pomocí abstrakce a spravovaly zdroje výpočetního prostředku.

V průběhu vývoje vzniklo několik různých typů operačních systémů. Základní rozlišení operačních systémů je možné provést z pohledu:

- jednouživatelské/víceuživatelské,
- reálného času,
- síťové.

2.3 Hlavní síťové operační systémy

Dominantní síťové operační systémy jsou UNIX a GNU/Linux.

Operační systém UNIX představuje označení pro celou rodinu operačních systémů. Operační systém lze označit jako UNIX po splnění sady testů. Charakteristickým rysem každého OS UNIX je jeho přehlednost. V případě OS UNIX lze použít tuto základní myšlenku – autor Doug McIllroy [16]:

„*Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*“[8].

Rozšířená množina vlastností, která vystihuje filozofické základy OS UNIX [28], je tato:

- Jednoduchost (KISS – keep it small and simple). U větších a složitějších nástrojů je vyšší pravděpodobnost chyby. Jednoduché nástroje jsou kombinovány tak, aby prováděly náročné úkoly.
- Znovupoužitelné komponenty v podobě knihoven.
- Standardní rozhraní pro definici vstupu a výstupu.

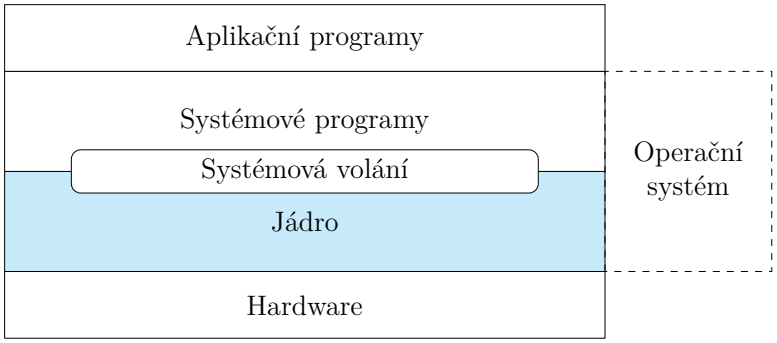
Linux je volně šiřitelné jádro (definice jádra je uvedena v následující kapitole) operačního systému. Linuxové jádro je dostupné se zdrojovými kódy. Pro vytvoření operačního systému se k jádru přidají další programy. **Linuxová distribuce** obsahuje jádro a další software. Tento přidaný software je typicky distribuován pod volnou licencí projektu GNU. Cílem GNU projektu je vyvinout operační systém se svobodnou licencí (bez kódu OS UNIX). Správné označení operačního systému je tedy GNU/Linux, což vychází z použitého jádra Linux a dalšího software pod licencí GNU. Běžně se však používá pro zkrácené označení Linux pro celý operační systém, které také budeme používat v tomto textu.

[8] Pište programy, které dělají jen jednu věc, ale dělají ji dobře. Pište programy, které pracují dohromady. Pište programy, které pracují s textovými proudy, protože ty představují univerzální rozhraní..

3 ARCHITEKTURA

3.1 Struktura operačních systémů

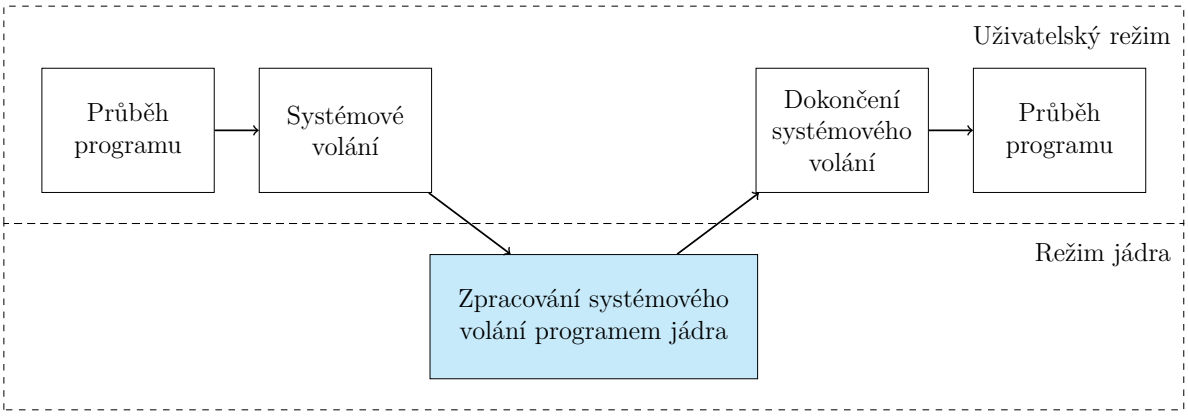
Typický operační systém obsahuje dvě části – **jádro** a **systémové programy**, viz obrázek 3.1. Jádrem nazýváme vše, co se nachází pod rozhraním systémových volání a nad fyzickými prostředky výpočetního prostředku. Zjednodušeně můžeme říci, že jádro je program, který pracuje přímo s hardware. Prostřednictvím **systémových volání** jádro poskytuje služby **systémovým programům**. Systémové programy poskytují prostředí pro spouštění **aplikačních programů**, které již nejsou typicky součástí operačního systému. Systémové programy mohou pouze zprostředkovávat rozhraní aplikačním programům pro využití systémových volání, nebo mohou poskytovat složitější funkce pomocí kombinace systémových volání, jako například ovládání souborů (vytvoření, mazání atd.). Systémové programy jsou typicky knihovny. Aplikační programy umožňují uživateli využívat prostředky zařízení. Aplikační programy nejsou typicky součástí operačního systému. Příkladem aplikací je textový editor nebo webový prohlížeč.



Obrázek 3.1: Struktura operačního systému.

Operační systém pracuje ve dvou základních režimech, **režim jádra** a **uživatelský režim**. Toto rozdělení není platné pro všechny systémy. Například dedikované systémy nemusí režimy rozlišovat a mají pouze privilegovaný režim. Režim jádra označujeme také jako privilegovaný/systémový režim nebo supervizor. Uživatelský režim označujeme také jako neprivilegovaný. Program běžící v privilegovaném režimu, tj. kernel a možné systémové programy, pracuje bez omezení. Programy v uživatelském režimu jsou omezovány tím, že procesor jim neumožňuje provést určité operace. Pokud program v neprivilegovaném režimu chce provést privilegovanou (chráněnou) operaci, musí „požádat“ program v privilegovaném režimu. Ten provede kontrolu akce a následně operaci provede. Jádro může delegovat práva určitým systémovým programům, které mohou provádět operace bez nutnosti volání jádra – pracují v privilegovaném režimu bez prostředníka (rychlejší provádění). Příkladem chráněných operací je alokace paměti, zakázaní přerušení, práce s I/O zařízeními. Obecně jsou to operace, které vyžadují koordinaci v operačním systému nebo které mohou ohrozit činnost OS, například chybou v programu. Chyba jádře/privilegovaném systémovém programu může vyřadit celý systém. Chyba v uživatelském režimu ovlivní chod pouze daného programu, systém by neměl být dotčen.

Při startu operačního systému se nejprve spustí program jádra, který běží v privilegovaném režimu. Další programy jsou startovány v uživatelském režimu. Uživatelský režim je změněn na režim jádra, když program volá systémové volání (vyžaduje chráněnou operaci) nebo pokud nastane přerušení (je potřeba přerušení obsloužit). Když program požádá o službu jádra prostřednictvím systémového volání, aktivuje se program jádra, který systémové volání obslouží, viz obrázek 3.2.



Obrázek 3.2: Volání služby jádra.

Přepnutí do režimu jádra se provádí pomocí instrukce instrukční sady architektury. V jazyce symbolických instrukcí je to *syscall* pro architekturu x86-64 nebo *swi* (software interrupt) pro architekturu ARM. Každá architektura má specifikováno, jak jsou předány vstupní parametry pro systémová volání tj. do kterých registrů. Tato specifikace je součástí binárního rozhraní ABI pro danou architekturu, viz kapitola 2.1. Bližší popis ABI pro jednotlivé architektury lze například najít v manuálových stránkách systémového volání *syscall* (`man syscall`).

I jednoduchý program v uživatelském režimu, ve kterém se v napsaném zdrojovém programu volá pouze jedno systémové volání, ve skutečnosti využívá chráněné služby jádra často. Ve výpisu 3.1 je zobrazen zdrojový kód programu, který se dotazuje na základní informace o systému. Tyto jsou dostupné pomocí systémového volání *uname*. Jednou z těchto informací je také architektura systému (*machine*). Po kompilaci zdrojového kódu `[]$ gcc sysCall.c -o sysCall` (parametrem -o se udává jméno výstupního binárního souboru) lze program spustit příkazem `[]$ ./sysCall`. Výstupem je pak označení architektury stroje, například x86_64.

```c
#include <stdio.h>
#include <sys/utsname.h>

int main(void)
{
 struct utsname info_str;
 uname(&info_str);
 printf("Architektura: %s \n", info_str.machine);
}
```

Výpis kódu 3.1: Program vypisující architekturu systému.

Skutečná volání jádra jsou uvedena ve výpisu 3.2, který zobrazuje pouze výběr. Jedná se o systémová volání *execve* – spuštění programu, *mmap* – alokace paměti, *open* – otevření souboru, *write* – v tomto případě výpis znaků do terminálu. S některými z těchto privilegovaných/chráněných funkcí se setkáme dále.

```
[]$ strace ./sysCall
execve("./sysCall")
mmap(NULL, 4096, MAP_ANONYMOUS)
open("/lib64/libc.so.6")
uname({sysname="Linux", nodename="PC-048D7K1", ...})
write("Architektura: x86_64 \n")
```

Výpis kódu 3.2: Zobrazení komunikace programu s jádrem.

U operačních systémů můžeme rozlišit tyto základní typy jejich stavby podle provedení jádra: monolitické systémy, systém klient-server, vrstvový systém a virtuální stroje [16].

3.1.1 Monolitické systémy a modulární jádro

Začněme poněkud zmatečně – strukturu **monolitického systému** můžeme definovat tak, že žádná struktura neexistuje. Jádro je napsáno jako souhrn procedur, které se navzájem volají. Každá z nich má přesně specifikované rozhraní (vstupní a výstupní parametry). **Monolitické jádro** vznikne kompilací všech procedur, resp. jejich zdrojových kódů, a následným svázáním vzniklých objektových modulů v sestavujícím programu (linker – linkage editor) do jednoho spustitelného souboru. Každá procedura je tak viditelná každé jiné. Pokud chceme nějakou funkci do jádra přidat, je nutno celé jádro znovu zkompilovat.

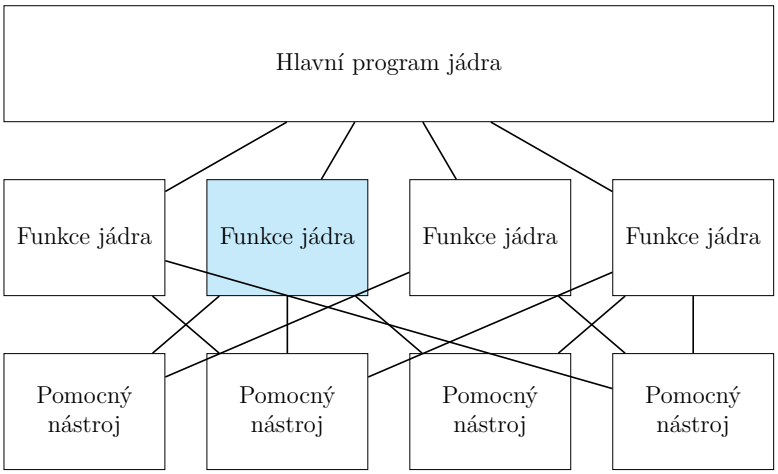
Navzdory tomu, že z globálního pohledu není zřejmá nějaká struktura, lze strukturu monolitického jádra popsat následovně:

- Hlavní program, který je spuštěn systémovým voláním (přerušením).
- Množina funkcí, které obslouží systémová volání (přerušení).
- Sada nástrojů, které pomáhají funkcím.

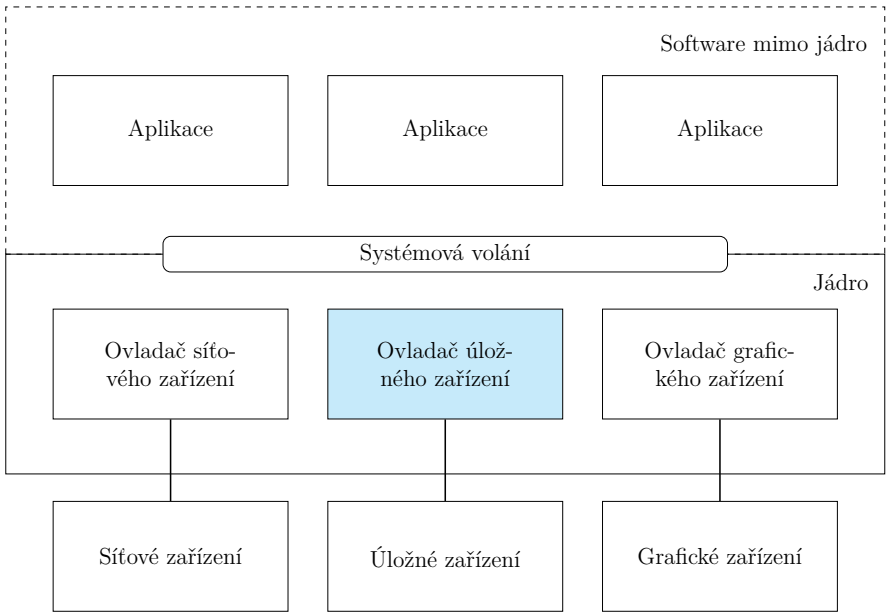
Hlavní program spouští funkci pro obsluhu systémového volání/přerušení. Každé syst. volání/přerušení obsluhuje jedna funkce. Úkolem pomocných nástrojů je provádět akce vyžadované několika funkcemi. Výsledný model je zobrazen na obrázku 3.3. Na obrázku jsou sice zobrazeny vrstvy, ale je nutno si jednotlivé prvky představit v ploše, protože vše je na stejné úrovni – není zde žádná hierarchie.

Monolitické jádro je rychlé, protože každá procedura volá přímo jinou. Problémem architektury je, že blokující chyba v programu jádra (například v ovladači) zablokuje celý systém. Na obrázku 3.4 je tato situace vysvětlena. Jádro je díky použitému rozhraní systémových volání chráněno proti chybám v uživatelských aplikacích. Pokud ale nastane chyba přímo v jádře, v ovladačích, nebo v jiných systémových službách, tak může dojít k pádu celého systému.

Monolitické jádro lze používat v modulární podobě – **modulární jádro**. V tomto případě jádro obsahuje pouze základní části pro běh. Další funkce jsou k dispozici v



Obrázek 3.3: Zjednodušená struktura monolitického jádra.



Obrázek 3.4: Monolitické jádro.

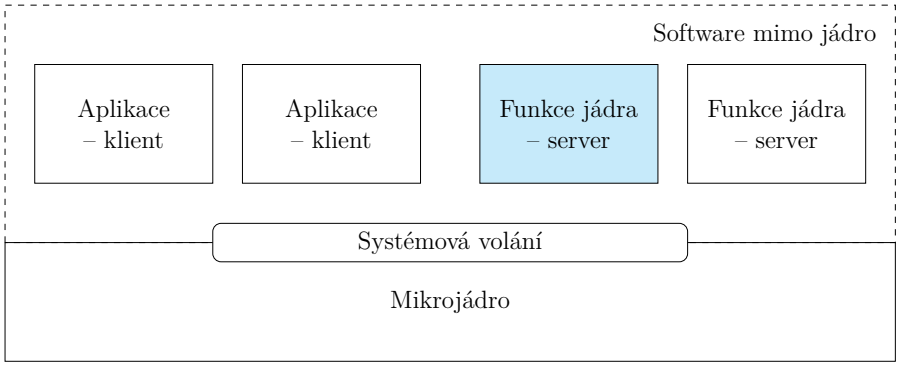
modulech, které jsou dynamicky načítány v případě potřeby. Moduly přináší výhodu v tom, že není nutná kompilace celého jádra, pokud jej potřebujeme rozšířit o nějakou funkci. Výhodná je také v menší velikosti jádra. Mezi moduly například patří ovladače zařízení a podpora souborových systémů.

Příkladem operačních systémů s monolitickým jádrem jsou BSD, UNIX System V, Linux, MS-DOS a FreeDOS.

3.1.2 Systémy klient-server, mikrojádro a hybridní jádro

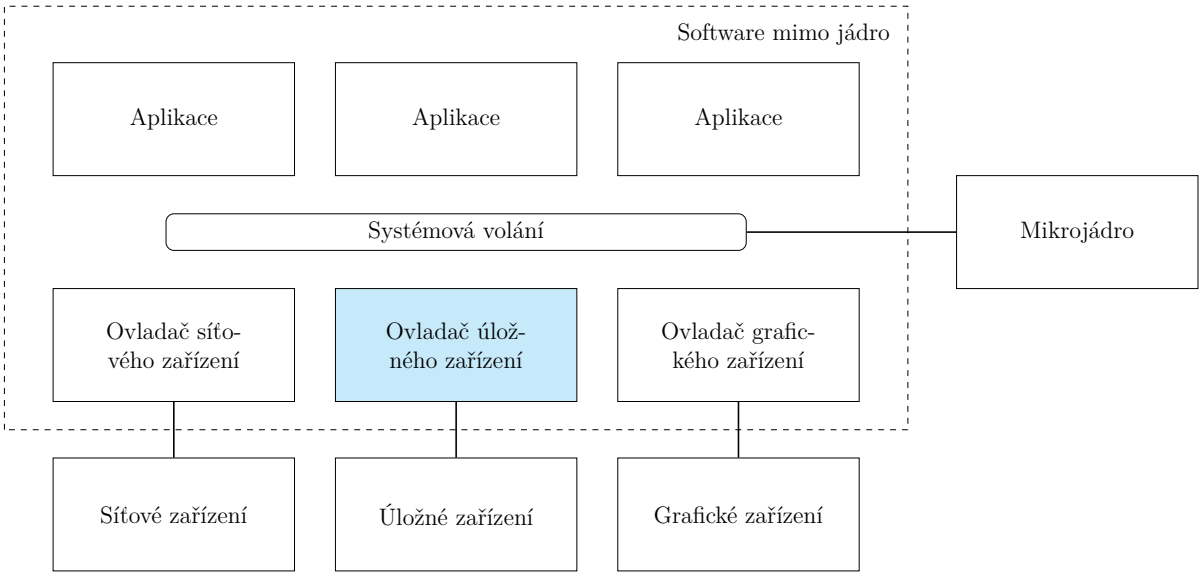
Systém klient-server vznikl přesunem funkcí z jádra do dalších programů – serverů, které běží v uživatelském režimu [11]. Jádro těmto programům přidělí částečná privilegovaná

práva pro přímý přístup do fyzické paměti DMA (Direct memory access). DMA umožňuje přímý přenos dat mezi fyzickou pamětí a vstupně/výstupními zařízeními bez účasti procesoru. Bez DMA jsou data přenášena přes registry procesoru. Servery typicky realizují ovladače síťových zařízení, zobrazovacích zařízení a souborových systémů. V jádře zůstává pouze nezbytné minimum – správa paměti, multitasking, přerušení, komunikace mezi procesy – a je nazýváno **mikrojádrem**. Systémové volání znamená vyslání požadavku programem v uživatelském režimu (klient) jinému programu v uživatelském režimu (server),viz obrázek 3.5.



Obrázek 3.5: Komunikace klient-server pomocí mikrojádra.

Servery vystupují v operačním systému jako systémové programy, viz obrázek 3.1. Chyba v kódu serveru způsobí pouze pád dané služby, nikoliv celého operačního systému, což je výhoda oproti monolitickým systémům. Na obrázku 3.6 si všimněte rozšíření oblasti, ve které chyba nezpůsobí zastavení činnosti jádra (a tudíž celého systému). Při chybě v serveru (např. ovladače) lze server restartovat a činnost tak obnovit, aniž by musel být restartován celý systém.



Obrázek 3.6: Mikrojádro.

Výhodou modelu klient-server je jeho snadná implementace pro distribuované systémy. Pokud klient komunikuje se serverem pomocí systémových volání v podobě zasílání zpráv, je jedno, zda odpověď klientu přichází z místní stanice nebo ze vzdálené. V případě distribuovaných systémů tedy určité síťové zařízení realizuje funkce jádra (server) pro jiná zařízení v síti (klienty). Klientská stanice má sice k dispozici vlastní jádro, ale to může fungovat pouze jako prostředek pro předávání zpráv v síti. Povšimněte si také rozdílu mezi konceptem monolitického jádra s dynamickými moduly, kdy moduly mezi sebou komunikují přímo. V tomto případě se samozřejmě jedná o výhodnější variantu.

Na druhou stranu má model klient-server určité nevýhody. Asi nejdůležitější je rychlost. Monolitické jádro je rychlejší. Pokud dále srovnáme mikrojádro s dynamickými moduly monolitického jádra, tyto moduly umožňují také vytvoření menšího jádra pouze se základními funkcemi. V tomto případě se jedná o lepší přístup, jelikož není potřeba zasílat zprávy jako u modelu klient-server. Příkladem operačních systémů s mikrojádrem je Hurd, Plan9, BeOS a QNX.

U modelu klient-server může být použito i **hybridní jádro**. Jedná se o mikrojádro, které obsahuje dodatečné funkce mimo základní. Tyto funkce jádra pracují rychleji, než pokud by byly implementovány v podobě serveru. Hybridní jádro je součástí operačního systému Windows [22].

3.1.3 Vrstvový systém a virtuální stroje

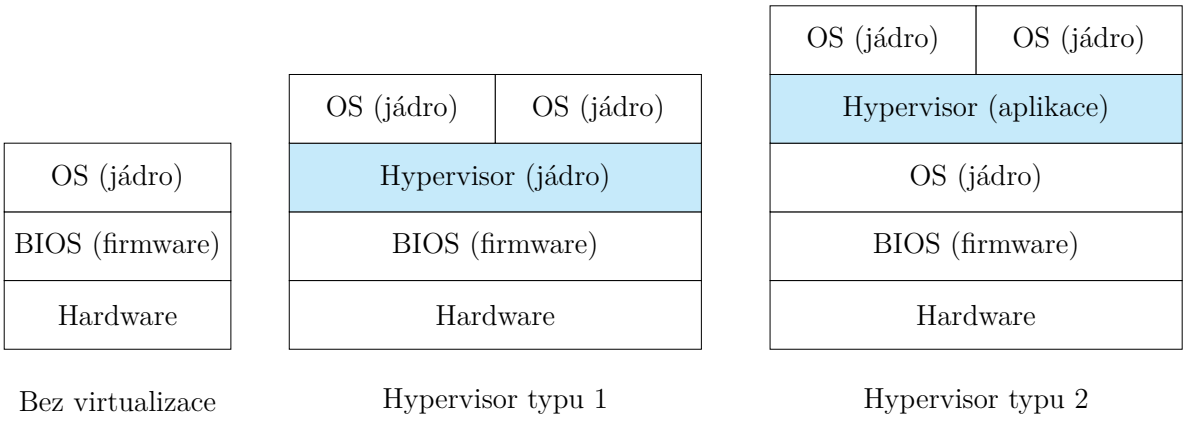
Principem tohoto přístupu je organizovat operační systém do **vrstev**. Vrstva je abstraktní objekt sestávající se z dat a operací manipulujících s těmito daty. Vrstva poskytuje služby pouze následující vyšší vrstvě a je implementována pouze pomocí funkcí její spodní vrstvy. Nejnižší vrstva je postavena nad hardware zařízení a nejvyšší vrstva je uživatelské rozhraní.

Zde je rozdíl oproti monolitickému systému, kde každý modul může komunikovat s každým. Dynamické moduly monolitického jádra se podobají vrstvovému přístupu v tom, že každá část jádra má přesně definované rozhraní.

Hlavní výhodou vrstvového přístupu je zjednodušení vývoje a snadné hledání chyb (služba dané vrstvy může být využita pouze přímo vyšší vrstvou). Další nevýhodou je pomalost, protože data musí být předávána přes všechny mezilehlé vrstvy. Problémem je složitá definice jednotlivých vrstev. Tyto mohou být například:

1. Přepínání mezi procesy
2. Organizace paměťového prostoru
3. Komunikace mezi procesy
4. Správa vstupních/výstupních zařízení
5. Uživatelské programy

Z vrstvových systémů vznikly **virtuální stroje** [16]. Na obrázku 3.7 je provedeno srovnání klasického přístupu: „hardware → BIOS → OS“ a virtuálních strojů: „hardware → BIOS → hypervisor → OS nebo hardware → BIOS → OS → hypervisor → OS“.



Obrázek 3.7: Virtuální stroje.

Hypervisor je program, který je vloženou vrstvou mezi hardware a virtuální stroj. Tato vrstva poskytuje abstrakci hardware (CPU, paměť, síťová rozhraní atd.) pro virtuální stroje. OS jako virtuální stroj pak domněle pracuje s celým hardware, tj. má například k dispozici svůj vlastní CPU a paměť. Jelikož je provedena abstrakce hardware, lze stejné virtuální stroje provozovat nad různým reálným hardware s jejich možným přesunem.

Hypervisor může být typu 1 nebo 2. Hypervisor typu 1 je situován přímo nad reálným hardware. Tento hypervisor je využíván pro provoz virtuálních strojů v datových centrech. Hypervisor typu 1 lze považovat za firmware s vlastním jádrem. Hypervisor typu 2 je situován v operačním systému a jedná se o aplikaci bez vlastního jádra. Tento hypervisor se běžně využívá na koncových stanicích. Virtuální stroj pro svůj běh vyžaduje práci v režimu jádra. V případě hypervisoru typu 2 je to však uživatelský režim hostitelského systému. Systémové volání ve virtuálním stoji tedy vyžaduje (několikanásobný) přesun mezi režimy pro dosažení režimu jádra hostitelského operačního systému. Hypervisor typu 1 umožňuje lepší využití prostředků hardware (z důvodu jeho přímého přístupu) než hypervisor typu 2.

Výhody virtualizace

- Hypervisor typu 1 – při poskytování síťový služeb lze zvýšit spolehlivost a využití hardware (úspora prostředků). Při řešení, kdy operační systémy poskytující službu běžící na jednom hardware, je při selhání části hardware tato služba nedostupná. V této konfiguraci dochází také k plýtvání prostředků, jelikož hardware je využíván pouze při poskytování služby (web, email atd.). Ke správě virtuálních strojů provozovaných nad hypervisorem typu 1 slouží tzv. „management console“. Management console umožňuje automatický přesun virtuálního stroje na jiný hardware v případě selhání původního, a tím v krátkém čase provést obnovu poskytování síťové služby. Management console také alokuje prostředky hardware pro virtuální stroje a tyto automaticky aktivuje v případě potřeby. Při alokaci lze provést i tzv. „over allocation“, tedy zabrání více prostředků hardware než je k dispozici. Této techniky se využívá pro provoz více virtuálních strojů, u kterých se předpokládá, že nikdy (nebo velmi zřídka) budou všechny plně využívat přidělené prostředky ve stejný čas.

- Hypervisor typu 2 – na koncových stanicích lze zvýšit bezpečnost pomocí oddělení virtuálních strojů. Přenos dat mezi operačními systémy lze řešit pomocí sdílení souborů na paměťovém úložišti nebo přes síť. Síťová komunikace může být řešena několika způsoby – může být vytvořena privátní síť, ve které jsou propojeny pouze operační systémy jednoho počítače. Virtuální stroje jsou také vhodné pro prostředí, kde je potřeba řešit nestandardní situace. Například se jedná o výukové laboratoře, kde lze provozovat upravené operační systémy pro různé předměty[1].

3.2 Příklady na jádro

Základním stavebním prvkem operačního systému je jeho jádro, které zprostředkovává komunikaci mezi uživatelskými procesy a hardware. Operační systém Linux využívá monolitické jádro. Na příkladu je ukázáno, které hlavní části toto jádro obsahuje.

Druhý příklad se zabývá důvody a způsoby kompilace jádra. Monolitické jádro rozšiřuje své funkce pomocí modulů. Tyto moduly mohou být nahrávány dynamicky za běhu operačního systému. Tímto způsobem se řeší problém s velikostí jádra a také velikostí jím zabrané paměti. V příkladu je ukázáno, kde se tyto moduly nachází. Je zde také popsán proces startu operačního systému.

V posledním příkladu je blíže vysvětlen pojem systémové volání. Pomocí systémových volání jádro zpřístupňuje své služby procesům.

3.2.1 Vlastnosti jádra

Nejdříve si připomeňme, že jádro je program, který pracuje přímo s hardware. Dále jsme si definovali, že monolitické jádro vznikne kompilací všech procedur, resp. jejich zdrojových kódů, a následným svázáním vzniklých objektových modulů do jednoho spustitelného souboru. Každá procedura jádra je přímo viditelná každé jiné. Ve vytvořeném souboru je tedy obraz jádra, který je nahrán při startu počítače.

V případě modulárního monolitického jádra jsou pomocí modulů doplňovány funkce jádra, které v něm nejsou obsažené tzn. jádro nebylo zkompilováno s těmito funkcemi. Tímto způsobem je možno jádru přidávat nové funkce dynamicky za běhu operačního systému. Dochází tím ke zmenšení velikosti jádra a úspoře zabrané paměti RAM.

V operačním systému Linux se modulární monolitické jádro nachází v adresáři `/boot`. Tento adresář obecně zahrnuje soubory potřebné pro start operačního systému. Výpis 3.3 zobrazuje část obsahu adresáře `/boot`. V souborech `vmlinuz*` jsou binární obrazy jádra Linux. V adresáři je typicky několik verzí jader, jelikož při aktualizaci jádra se ponechávají v systému starší verze pro případ, že by nová verze jádra nepracovala správně. Při startu operačního systému lze ovlivnit, jaké jádro bude použito. Často je pro soubor s obrazem jádra použit formát ELF (Executable and Linkable Format), který zajišťuje přenositelnost binárního kódu. Proto není jádro nutné kompilovat při jeho použití mezi stejnými systémy (stejná architektura CPU).

```
1 []$ ls /boot/
```

[1] První použití na fakultě v roce 2007 pro laboratoř Cisco akademie.

```
config-verze1.distribuce.x86_64
config-verze2.distribuce.x86_64
initramfs-verze1.distribuce.x86_64.img
initramfs-verze2.distribuce.x86_64.img
vmlinuz-verze1.distribuce.x86_64
vmlinuz-verze2.distribuce.x86_64
```

Výpis kódu 3.3: Soubor jádra a s ním svázané soubory.

Ve výpisu 3.3 jsou také další soubory přímo svázané s jádrem. Jedná se o dočasný souborový systém s moduly jádra potřebnými pro start operačního systému – soubor `initramfs*`. Důvod použití tohoto souborového systému je v redukci velikosti souboru s jádrem. Distribuce Linuxu typicky obsahují obecné jádro, které neobsahuje všechny ovladače potřebné pro start na všech možných zařízeních (bylo by příliš velké). Po detekci použitého hardware se potřebné moduly nahrají do jádra a systém nastartuje.

Aktuálně používané moduly v jádře lze získat příkazem `lsmod`, jak je zobrazeno ve výpisu 3.4. Po startu jsou další moduly jádra pro činnost systému k dispozici na připojeném paměťovém úložišti, typicky v adresáři `/lib/modules/<verze_jadra>/`. Informace o umístění souboru se specifickým modulem lze získat příkazem `modinfo nazev_modulu`.

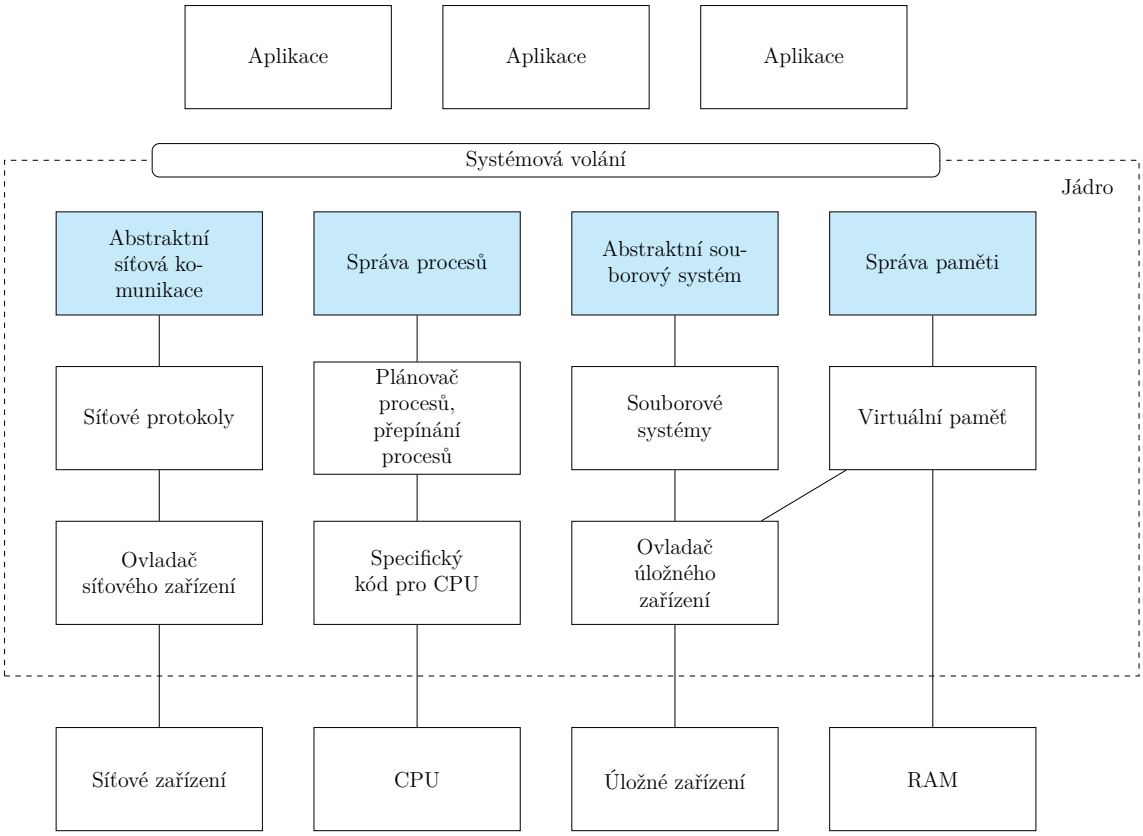
```
[]$ lsmod
Module          Size
fat             65913
radeon          1596647
usb_storage     66523
bluetooth       372944
...
```

Výpis kódu 3.4: Seznam modulů nahraných do jádra.

Konfigurace použitá při kompilaci jádra, tj. s jakými moduly a částmi bylo jádro vytvořeno, je uložena v dalším souboru – `config*`. Tento soubor slouží pro zjištění, s jakými parametry bylo jádro kompilováno a které moduly jsou přímo v jádře nebo jsou dostupné pro dynamické použití (blíže k modulům viz dále). Pro každé jádro jsou v adresáři zvláštní soubory.

Příklad struktury monolitického jádra je zobrazen na obrázku 3.8 [13]. Programy v uživatelském režimu využívají pro svou činnost systémová volání. Rozhraní pro systémová volání na obrázku tvoří horní část jádra. Pod tímto rozhraním se pak nachází správa systémových prostředků. Jedná se o správu abstraktní síťové komunikace, která umožňuje použití různých komunikačních protokolů. Typicky se jedná o protokoly TCP/IP. Komunikační protokoly pak využívají služeb ovladače konkrétního síťového rozhraní. Správa procesů má za úkol vytváření a ukončování procesů, organizaci jejich vykonávání na CPU a řešení meziprocesorové komunikace. Plánovač procesů řeší přechody mezi stavy činnosti procesů (proces může být například ve stavu čekání, nebo být vykonáván na CPU). Úkolem plánovače je také implementace priorit při zpracování procesů. Použití různých typů CPU řeší kód, který je specificky závislý na použité architektuře. Z tohoto důvodu také existují verze zkompilovaných jader pro různé hardwarové architektury. Abstraktní souborový systém sjednocuje používání různých souborových systémů, jako například Ext4

a FAT32. Souborové systémy jsou v jádře typicky podporovány v podobě přídavných modulů – jádro tak nemusí obsahovat ovladače všech souborových systémů. Souborové systémy pracují s ovladačem konkrétního úložného zařízení. Správa paměti řeší přidělování paměti procesům. Virtuální paměť kombinuje paměť RAM a prostor na úložném zařízení (viz propoj na obrázku mezi blokem virtuální paměti a ovladačem úložného zařízení). Virtuální paměť také řeší přesuny dat mezi RAM a prostorem na disku.



Obrázek 3.8: Základní části monolitického jádra.

3.2.2 Kompilace jádra

Jádro Linux lze vytvořit (zkompilovat) ze zdrojových kódů, které lze volně získat na Internetu. Zdrojové kódy jsou k dispozici pro různé verze a varianty jader.

Originální jádro, které není vázáno na žádnou distribuci, se nazývá vanilla (bez příchuti). Vanilla jádro je k dispozici na adrese *kernel.org*. Toto jádro je upraveno podle požadavků (účelu) dané distribuce na **distribuční** jádro. Tvůrci distribuce mohou k jádru i přidat svůj kód a jádro zkompilují podle zvolené konfigurace. Pro jednoduchou distribuci jádra vytvoří softwarové balíčky, například typu RMP nebo DEB (přípona `.rpm` | `.deb`), kde je jádro k dispozici v binární nebo zdrojové podobě. Přístup ke zdrojovým kódům jádra umožňuje provést další změny jádra pomocí vlastní kompilace.

Proč kompilovat jádro? Původně byl hlavní důvod komplikace jádra pro podporu systémové architektury. V dnešní době jsou pro distribuce k dispozici kompilovaná jádra

pro různé architektury, např. x86_64, ARM64 (aarch64) a IBM Power (ppc64le). Jádra distribucí jsou také kompilována s univerzální konfigurací, aby pracovala co s nejširší hardwarovou výbavou. Pro běžné použití tak kompilace jádra není potřeba.

Jádro je potřeba kompilovat při použití na systémech se specifickou výbavou, omezeným hardware nebo potřebě specifických funkcí jádra. Jedná se například o podporu dalších periferií, podpora činnosti „real-time“ nebo bezpečnostní rozšíření. Typickým příkladem pro kompilaci je zmenšení jádra z důvodu omezeného prostoru na paměťovém úložišti a omezené velikosti paměti RAM. Zabrané místo na paměťovém úložišti je dáno velikostí (komprimovaného) obrazu jádra. Dále může být přítomen soubor se základními ovladači (jako obraz virtuálního disku). V paměti RAM pak jádro zabírá místo odpovídající jeho nekomprimované velikosti, tj. po jeho spuštění. Další místo v paměti je pak zabráno dynamicky alokovanými daty (halda, zásobník, blíže viz kapitola 4).

Komprimované jádro v univerzální distribuci CentOS má velikost okolo 6 MiB – soubor `vmlinuz` v adresáři `/boot`, viz výpis 3.5. Znak *z* v názvu souboru značí, že se jedná o komprimovaný soubor (znaky *vm* označují použití konceptu virtuální paměti). Moduly jádra, které jsou nutné pro start systému, jsou uloženy v podobě obrazu (image) virtuálního disku (v RAM) (soubor `initramfs`), který má velikost okolo 30 MiB. Tento virtuální disk obsahuje ovladače zařízení pro připojení souborového systému na paměťovém úložišti. V připojeném souborovém systému jsou pak další moduly jádra v adresáři `/lib/modules`. Minimální konfigurace jádra podporující IDE (řadič pro připojení disku), ext2 (souborový systém na disku), TCP/IP, NIC (síťová karta) může mít velikost pouze okolo 400 KiB (Linux Tiny). Existují i minimalizované distribuce (Tiny Core Linux).

```
[]$ ls -sh /boot/
5,2M vmlinuz-verze.distribuce.x86_64
5,2M vmlinuz-verze.distribuce.x86_64
...
32M  initramfs-verze.distribuce.x86_64.img
32M  initramfs-verze.distribuce.x86_64.img
```

Výpis kódu 3.5: Velikost jádra a obrazu virtuálního disku s moduly.

Co bude obsahovat nové jádro lze zvolit při konfiguraci kompilace. Pro konfiguraci jádra existuje několik nástrojů v textovém nebo grafickém režimu (`make gconfig` nebo `make menuconfig`). U jednotlivých částí jádra lze zvolit tyto možnosti:

1. nekompilovat (nezačleňovat do jádra),
2. kompilovat přímo do jádra,
3. kompilovat ve formě modulu, který je zaveden v případě potřeby.

Konfigurace jádra je uložena v souboru `.config`. Základní možnosti při konfiguraci jádra jsou:

- Loadable Module Support – Jedná se o obecnou volbu, která musí být povolena pro možnost použití dynamických modulů.
- Processor Type and Features – Volba typu procesoru je jeden ze základních parametrů kompilace. Nahlédnutím do souboru `/proc/cpuinfo` lze zjistit, jaký typ

procesoru je použit. V této volbě lze také určit typ přepínání procesů – Preemptible kernel, viz dále preemptivní a kooperativní přepínání procesů.

- Device Drivers – NIC, grafická karta
- Podpora sítě – TCP/IP
- File Systems – Ext4, XFS, SMB
- Security Options – SELinux

Při volbě konfigurace jádra je potřeba mít na paměti, které vlastnosti jádra jsou nutné pro daný HW a činnost. Při nevhodné kombinaci překlad jádra selže, nebo jádro přeložit půjde, ale po zavedení nebude pracovat. Například z důvodu, že nebude podporován souborový systém, který je použit na úložném zařízení.

Po překladu pomocí nástroje `make` vznikne binární soubor jádra a soubory modulů. Při instalaci je soubor s jádrem umístěn do adresáře `/boot`. Zde je také umístěn konfigurační soubor jádra, aby bylo možno zpětně zjistit, s jakou konfigurací (s jakými vlastnosti) bylo jádro zkompilováno. Přeložené moduly jádra jsou umístěny do adresáře `/lib/modules` (připojený souborový systém).

Instalace jádra nepřepisuje původní jádro. V případě nefunkčnosti nového jádra tak zůstává k dispozici funkční jádro. Typicky je udržováno několik posledních verzí jader, například 5. Při startu může být zobrazena možnost volby, které jádro bude zavedeno.

Shrnutý postup kompilace a zavedení nového jádra je následující:

1. Stažení zdrojových kódů, například pomocí balíčku pro danou distribuci.
2. Konfigurace jádra.
3. Překlad jádra.
4. Instalace jádra do příslušných adresářů a úprava k zavaděče systému pro výběr nového jádra.
5. Restart a zavedení nového jádra.

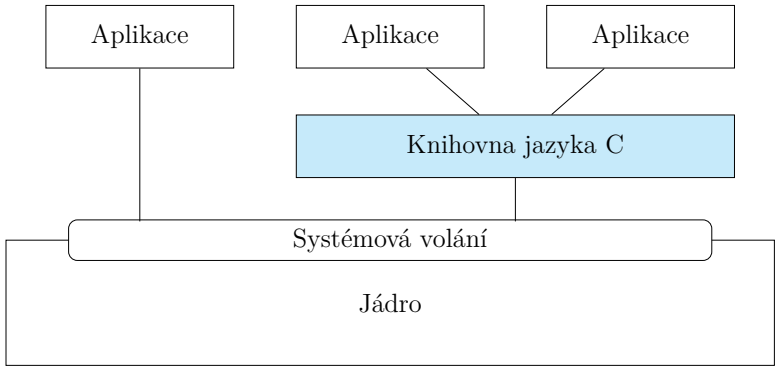
3.2.3 Systémové volání

Pro popis příkladu realizace systémových volání si zopakujme:

- Jádro poskytuje služby systémovým programům prostřednictvím systémových volání.
- Systémové programy umožňují běh aplikačních programů.
- Systémové programy mohou poskytovat složitější funkce, jako například ovládání souborů (vytvoření, mazání atd.).

Systémová volání pro služby jádra lze volat přímo nebo pomocí prostředníka – knihovny. V OS Linux je to *GNU C Library* (`glibc`). Použití knihovny jako prostředníka přináší univerzálnost tím, že převádí systémová volání poskytované konkrétním jádrem na standardní funkce jazyka C a C++, které jsou tak nezávislé na jádře. Tato knihovna podporuje různá jádra, jako například Linux, Hurd, FreeBSD, a NetBSD. Pozice knihovny je zobrazena na obrázku 3.9 [17]. Zdůrazněme, že knihovna není součástí jádra, ale pracuje

v uživatelském režimu. Knihovnu můžeme zařadit dle smyslu obrázku 3.1 do kategorie systémových programů. Budeme tedy dále v textu používat slova „funkce knihovny“ a „volání jádra“ jako vzájemně záměnná.



Obrázek 3.9: Příklad použití knihovny pro volání systémových volání jádra.

Existují i alternativní knihovny poskytující nezávislost funkcí pro systémová volání na konkrétním jádře. Tyto knihovny se odlišují svou velikostí (jsou typicky menší) a podporou jader. Příkladem je knihovna *uClibc*, jejíž primární nasazení je pro malá mobilní a dedikovaná zařízení.

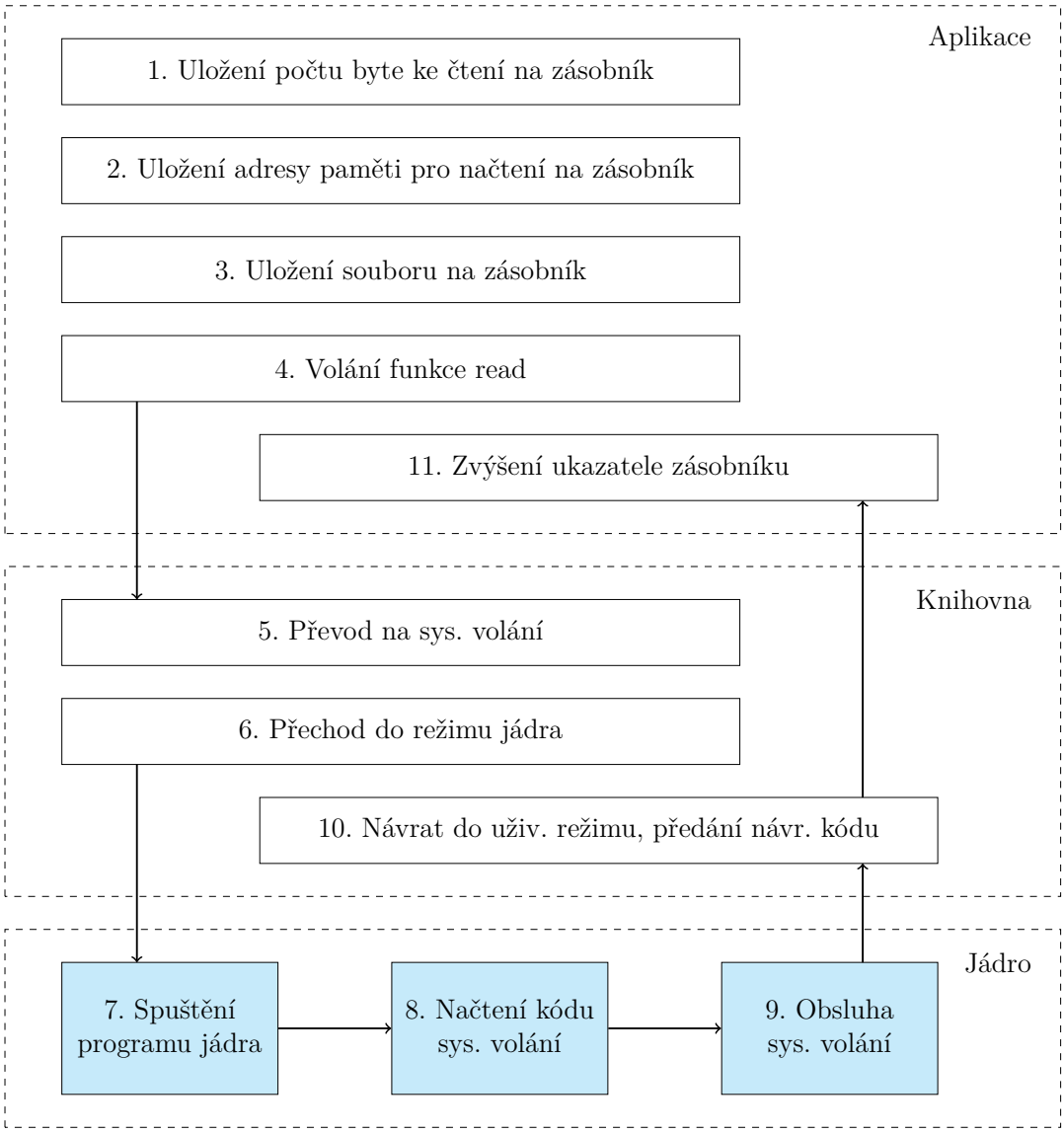
Systémová volání jsou prováděna v posloupnostech kroků. Obecný postup je následující:

1. Předání parametrů systémového volání do registrů.
2. Přepnutí do režimu jádra.
3. Obsluha systémového volání.
4. Návrat do uživatelského režimu.
5. Vrácení hodnot a návratového kódu.

Uveďme příklad systémového volání a funkce knihovny jako prostředníka pro čtení dat ze souboru [11]. Použita bude funkce knihovny s názvem *read* s těmito třemi parametry: soubor, vyrovnávací paměť (buffer) a počet bajtů ke čtení.

Před voláním knihovní funkce *read* program nejdříve uloží parametry na zásobník[2]. Názorněji viz kroky 1 až 3 na obrázku 3.10. Kompilátory jazyků C a C++ ukládají parametry v reverzním pořadí. První a třetí parametr jsou hodnoty, druhý je odkaz. Krok 4 pokračuje voláním knihovní funkce.

[2]Funkce zásobníku bude popsána později.



Obrázek 3.10: Kroky pro čtení dat ze souboru pomocí knihovní funkce a systémového volání.

Knihovna převede funkci *read* na systémové volání – uloží do registrů procesoru kód systémové volání a jeho parametry (krok 5). Následně knihovna vyvolá softwarové přerušení (krok 6)[3] a spustí tak program jádra (krok 7). Program jádra si načte z registrů uložené hodnoty a obslouží systémové volání. Po skončení programu jádra se řízení vrací do knihovní funkce v uživatelském režimu (krok 10). Tato funkce následně předá návratové hodnoty a návratový kód, tj. informaci o tom, jak systémové volání proběhlo – úspěšně či neúspěšně s číslem chyby, viz příklady ve výpisu 3.6. K ukončení práce (krok 11) je ještě zapotřebí vyčistit zásobník, tzn. zvýšit hodnotu ukazatele zásobníku na hodnotu, kterou měl před uložením parametrů funkce *read* pro čtení dat ze souboru.

[3]Název instrukce architektury pro vyvolání softwarového přerušení se liší. Z historických důvodů může být celá tato akce označována jako „trap“, protože název této instrukce v jazyce symbolických instrukcí byl *trap*. Typický název instrukce je také „INT“, která se používá v architektuře x86.

```
/*File name too long*/
#define ENAMETOOLONG 36
/*Function not implemented*/
#define ENOSYS       38
/*Directory not empty*/
#define ENOTEMPTY    39
```

Výpis kódu 3.6: Návratový kód – číslo případné chyby při provádění systémového volání.

4 PROCESY

4.1 Definice procesu

Operační systém pracuje s procesy a přiděluje jim systémové zdroje. Pod pojmem proces si lze představit program, který je realizován operačním systémem. Jeden program může být realizován několika procesy. Komunikace dvou počítačů v síti je zprostředkována pomocí komunikace mezi procesy, které běží na těchto počítačích.

Operační systémy mohou označovat procesy celočíselným **identifikátorem** PID (Process IDentifier). Tuto celočíselnou hodnotu přiděluje jádro v okamžiku vytvoření procesu. PID se přiděluje lineárně, a lze tedy určit časovou souslednost vytvoření procesů, jak je zobrazeno ve výpisu 4.1. PID nemá typicky vypovídající hodnotu o činnosti a druhu procesu, až na výjimky[1]. Proces s $PID = 1$ může mít v systému výstavní postavení a nelze jej ukončit (resp. je vždy systémem znovu spuštěn). Jedná se například o proces *systemd* (dříve *init*).

```
[]$ ps ax | head
PID  TTY  STAT  TIME  COMMAND
1    ?    Ss    0:30  /usr/lib/systemd/systemd
2    ?    S     0:00  [kthreadd]
4    ?    S<    0:00  [kworker]
6    ?    S     0:02  [ksoftirqd]
7    ?    S     0:00  [migration]
```

Výpis kódu 4.1: Časová posloupnost vzniku procesů.

Operační systémy mohou procesy organizovat do stromové struktury, jak je zobrazeno ve výpisu 4.2. Hodnota PID je zde uvedena v závorce za názvem procesu. Proces s $PID = 1$ tvoří vrchol stromové hierarchie procesů.

```
[]$ pstree -p | head
systemd(1)-+-ModemManager(907)-+-{ModemManager}(944)
           |                   `-{ModemManager}(951)
           |-NetworkManager(1069)-+-dhclient(1236)
           |                      |-{NetworkManager}(1077)
           |                      `-{NetworkManager}(1082)
```

Výpis kódu 4.2: Stromová struktura procesů a jejich PID.

Dříve operační systémy pracovaly pouze s jedním procesem v daném čase. Tento proces měl přístup ke všem prostředkům. V dnešní době operační systémy umožňují „současný“ běh[2] více procesů a tyto procesy „soupeří“ o prostředky výpočetního stroje.

[1] V OS Linux se proces s $PID = 0$ nazývá vyměňovací (swapper) a je zodpovědný za činnost virtuální paměti. Tomuto procesu neodpovídá žádný program. Pracuje pouze v režimu jádra a není proto vidět ani ve výpisu procesů v uživatelském režimu. Je jediným procesem, pro který neplatí, že je vytvořen voláním jádra *fork*, viz dále.

[2] Nezaměňujme za vykonávání, jeden procesor může v jednom čase vykonávat pouze jeden proces.

Procesy můžeme rozdělit na **systémové** a **uživatelské**. Systémové procesy jsou vytvořeny systémovými programy a jádrem. Uživatelské procesy jsou vytvořeny uživatelskými programy.

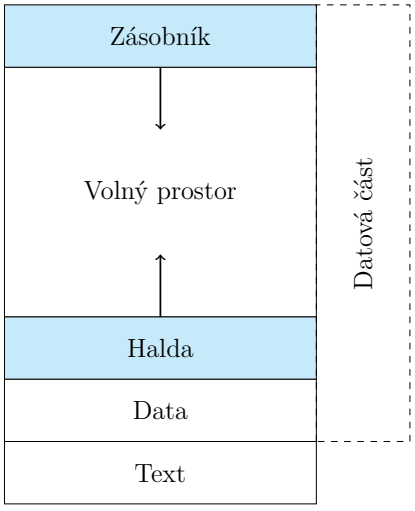
Informace o procesech jsou uloženy v jejich kontextech. V **uživatelském kontextu** jsou informace, které proces potřebuje pro vlastní činnost, například jeho programový kód. V **kontextu jádra** jsou udržovány informace, které potřebuje operační systém, aby mohl proces spravovat. Jedná se například o hodnotu programového čítače. Více procesů (vláken, viz dále) může sdílet stejný kontext nebo jeho části.

Přepnutí kontextu se provede, když je vykonáván jiný proces. Jádro udržuje v kontextu jádra poslední stav vykonávání původního procesu, takže později může jeho činnost obnovit tam, kde byl přerušen.

4.2 Struktura procesu

4.2.1 Uživatelský kontext

Uživatelský kontext je tvořen z několika částí, které mohou mít pevnou velikost (statické), nebo se jejich velikost může měnit při běhu procesu (dynamické), viz obrázek 4.1 [11]. Jedná se o **textovou část** (text segment), která obsahuje programové instrukce. Přístup k této části může být pouze pro čtení, čímž se zablokuje možnost modifikace programových instrukcí. Díky tomu může více procesů téhož programu jednoduše sdílet textovou část. **Datová část** (data segment) obsahuje data, která proces potřebuje již od svého startu. Jedná se o globální proměnné, řetězce, datová pole a další. **Halda** (heap) je použita, když proces za běhu dynamicky alokuje datový prostor pro data, která potřebuje až při svém běhu. **Zásobník** (stack) je také dynamicky alokovaný, ovšem zde se ukládají parametry procesem volaných funkcí. Je obvyklé, že zásobník roste směrem k nižším adresám, tedy tzv. vrchol zásobníku se pohybuje směrem dolů. Jedná se o zásobník typu LIFO (Last-in First-out). Mezera mezi haldou a zásobníkem představuje volné místo, které dovoluje dynamickou změnu zásobníku a haldy za běhu procesu.

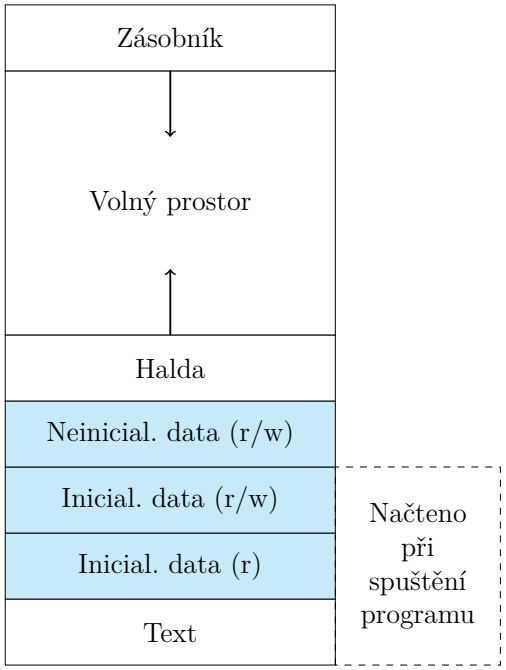


Obrázek 4.1: Uživatelský kontext procesu – statické a dynamické části.

Při použití virtuální paměti může aktuální velikost uživatelského kontextu procesu přesáhnout velikost dostupné paměti RAM.

Na obrázku 4.2 je provedeno podrobnější rozdělení datové části procesu na:

- Inicializovaná data dostupná pouze pro čtení – datové prvky inicializované programem, které nelze měnit.
- Inicializovaná data dostupná pro čtení i zápis – datové prvky inicializované programem, lze měnit.
- Neinicializovaná data[3] – datové prvky, které program neinicializoval, lze měnit.



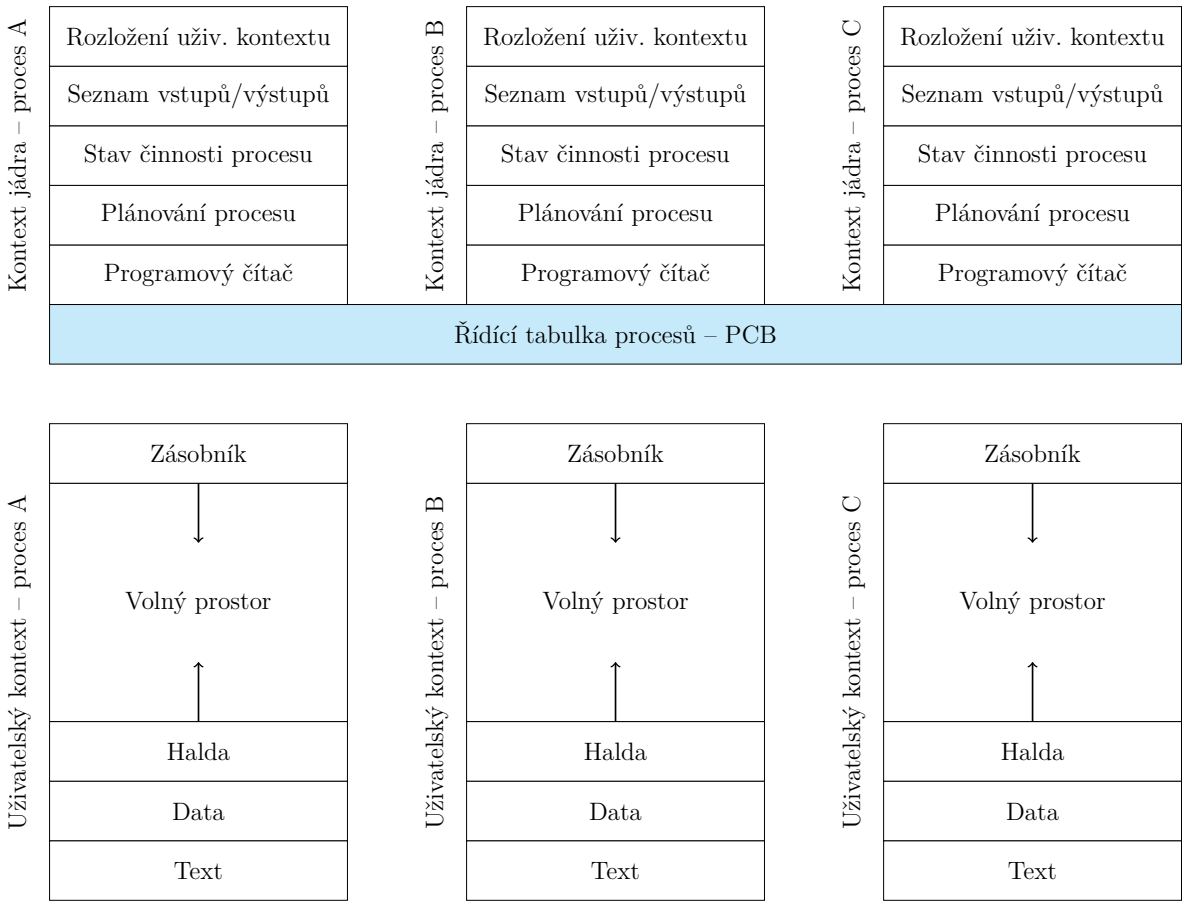
Obrázek 4.2: Datová část paměti procesu.

Rozdělení datové části procesu na inicializovaná a neinicializovaná data je provedeno pro zmenšení velikosti souboru s programem – soubor je menší o velikost neinicializovaných dat. V souboru je uveden pouze počet bajtů pro alokaci v paměti při spuštění programu. Jejich počáteční hodnota může být náhodná nebo nastavena operačním systémem na nulu, v případě ukazatele na „null“. Ovšem použití těchto dat nemusí být žádoucí ve všech případech. Mikrokontroléry mohou mít velmi malou paměť RAM (např. 512 B) a relativní dostatek paměti FLASH (např. 4 KiB). Zde je žádoucí program sestavit tak, aby neinicializovaná data měla co nejmenší velikost (tedy naopak zvětšit velikost spustitelného souboru).

[3] Z historických důvodů se jim říká zkráceně BSS (Block Started by Symbol), což byla instrukce počítače IBM 7090.

4.2.2 Kontext jádra

Informace v kontextu jádra poskytují údaje pro správu běhu procesů. Kontexty jádra se mohou nacházet v **řídící tabulce procesů** – PCB (Process Control Block), viz obrázek 4.3 [16]. Tato tabulka obsahuje informace pro všechny procesy, které jsou spuštěny. Identifikátor PID funguje v tabulce jako index pro daný proces.



Obrázek 4.3: Řídící tabulka procesů.

Kontext jádra procesu udržuje informace o rozložení jeho uživatelského kontextu. Dále udržuje informace o jeho správě operačním systémem, kterými například jsou:

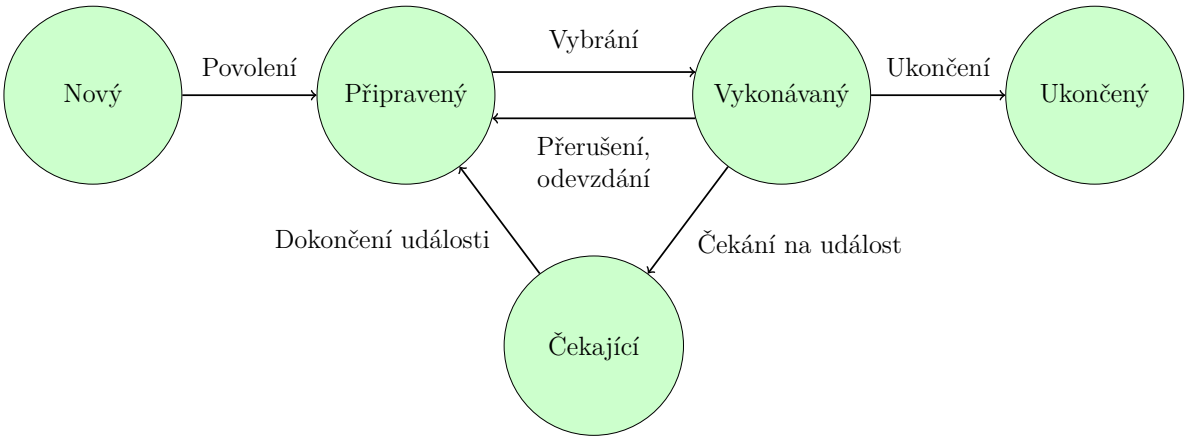
- Programový čítač obsahuje adresu následující instrukce strojového kódu, která bude vykonána procesorem.
- Parametry spojené s plánováním běhu procesů – priorita procesu, čas aktivity v CPU, plánovací fronta atd. Tyto informace jsou použity při plánování procesů, tj. rozhodování, který proces bude spuštěn jako další.
- Seznam vstupů/výstupů, které proces používá. Jedná se například o otevřené soubory.
- Stav činnosti procesu.

Poslední uvedená informace obsahuje záznam o stavu běhu procesu. Proces se může nacházet v jednom z několika možných základních stavů:

1. Nový – proces je vytvářený.
2. Připravený – proces čeká na procesor.
3. Vykonávaný – instrukce jsou prováděny.
4. Čekající – proces čeká na nějakou událost, například čtení dat z paměťového média.
5. Ukončený – proces dokončil vykonávání svých instrukcí.

Jeden proces může být ve stavu „vykonávaný“ na jednom procesoru a více procesů se může nacházet například ve stavu „připravený“ a „čekající“.

Přechody mezi těmito stavy procesu jsou zobrazeny na obrázku 4.4 [16, 30]. Poté, co je proces vytvořen, přechází do stavu „připravený“ ke zpracování. Až plánovač procesů proces vybere, je proces vykonáván. Vykonávání může být ukončeno poté, co proces dobrovolně opustí procesor, nebo v případě příchozího přerušení, nebo v případě zablokování procesu z důvodu čekání na určitou událost (například na data, která se načítají). V případě zablokování se proces po ukončení čekání na požadovanou událost dostane do stavu „připravený“. Nyní proces čeká, až je vybrán a přechází do stavu „vykonávaný“. Jak proces končí svou činnost, přechází ze stavu vykonávaný do stavu „ukončený“. Do stavu „ukončený“ se lze dostat jenom ze stavu „vykonávaný“, protože ukončení se musí vykonat. Při přepnutí kontextu se aktuální stav běhu procesu uloží v řídící tabulce procesů. Dále se načte poslední stav běhu nového procesu.

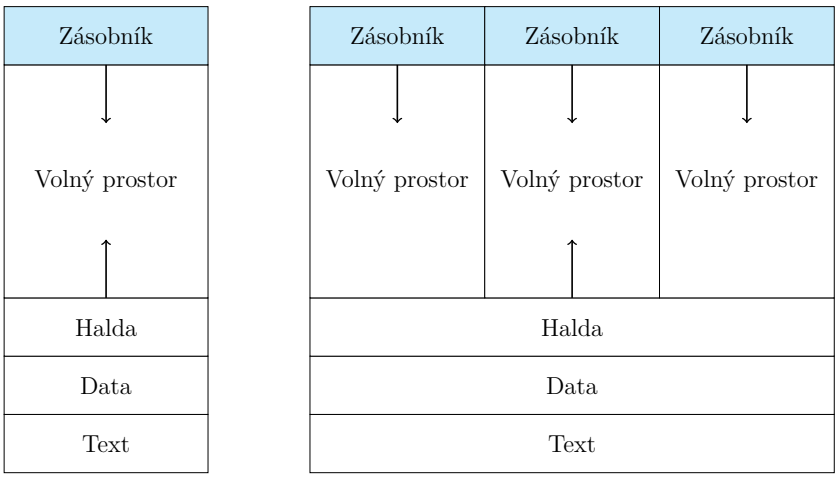


Obrázek 4.4: Základní stavy běhu procesu.

4.3 Vlákna

Vlákno je základní prvek, který je vykonávaný procesorem. Proces může mít více vláken. Jedno vlákno umožňuje provádět jeden úkol v daném čase. Více vláken může provádět stejný úkol s různými daty (viz dále příklad webového serveru). Aby úkol nad různými daty mohl být prováděn paralelně, musí systém disponovat více procesory nebo alespoň jedním

vícejádrovým procesorem[4]. Jeden procesor nebo jedno jádro procesoru vykonává jedno vlákno[5]. Záleží také na použitém modelu realizace vláken v operačním systému, viz dále. Návaznost uživatelského kontextu procesu na jeho vlákna je zobrazena na obrázku 4.5. Každé vlákno obsahuje vlastní zásobník. Vlákna sdílí haldu[6], programový kód a datovou oblast.



Obrázek 4.5: Uživatelský kontext procesu – jednovláknový a vícevláknový.

Příkladem použití vláken je webový server. Server obsluhuje požadavky velkého počtu klientů. Řešení bez použití vláken je, že pro nový požadavek se spustí další proces, který klienta obslouží. Takto nově vytvořený proces bude vykonávat stejnou činnost jako proces, který již existuje. Toto řešení přináší režii, která je dána vytvářením několika procesů vykonávajících stejnou práci. Při použití vláken je pro obsloužení příchozího požadavku spuštěno nové vlákno stejného procesu [16].

Výhody vícevláknového přístupu lze shrnout následovně:

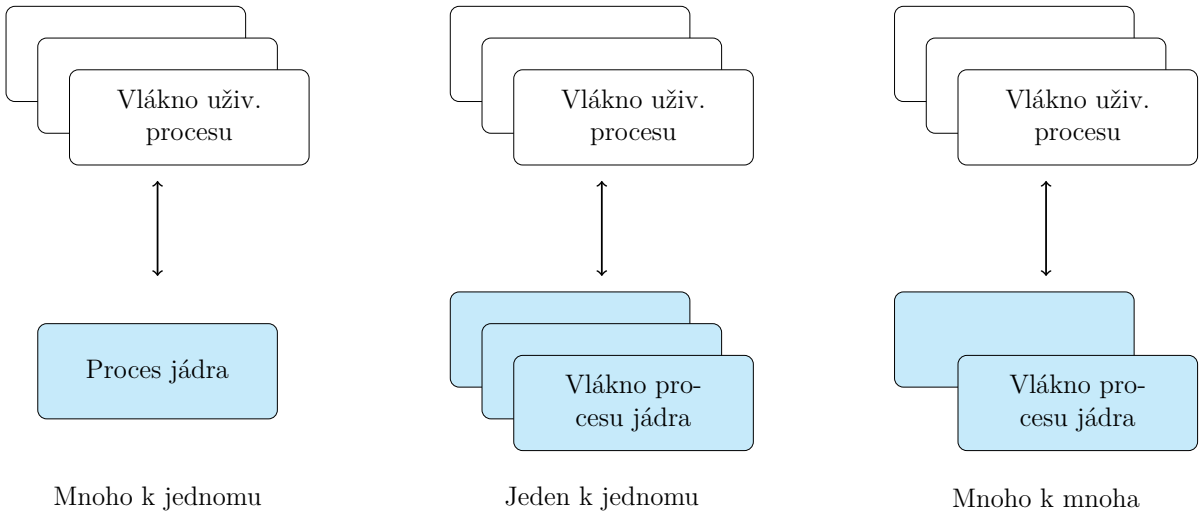
- Rychlost odezvy. Umožňuje pokračování vláken procesu pokud nějaké vlákno vyžaduje delší dobu zpracování.
- Sdílení prostředků. Vlákna sdílí většinu částí procesu a není tak zabíráno tolik paměti.
- Hospodárnost. Správa vláken je z pohledu režie lepší než správa samostatných procesů. Vytvoření procesu je delší než vytvoření vlákna.

Vlákna přináší i několik nevýhod. Sdílení paměťového prostoru mezi vlákny může způsobit nekonzistentnost dat. Například jedno vlákno čte proměnnou, zatímco jiné vlákno tuto proměnnou zapisuje. Tato situace je nazývána souběh a je potřeba pro tento případ činnost vláken synchronizovat. Programový kód vlákna má tedy zvýšenou složitost v tom, že programátor musí na tyto stavy pamatovat. Neřešená práce se sdílenými daty může také vést k uvíznutí systému. Bližší informace k synchronizaci a souběhu jsou uvedeny v kapitole 4.7. Problém uvíznutí je popsán blíže v kapitole 4.8. Další nevýhodou je, že

[4]Jedna komponenta obsahující více CPU.
[5]Pokud není použitý tzv. hyperthreading – z fyzického CPU jsou vytvářeny logické CPU.
[6]V některých operačních systémech mají vlákna i vlastní haldu.

chyba v jednom vlákně může způsobit pád celého procesu, tedy i ostatních vláken téhož procesu.

Z pohledu správy OS se rozlišují dva typy vláken. **Vlákna jádra** náleží procesu jádra[7]. **Uživatelská vlákna** náleží procesům mimo jádro. Mezi uživatelskými vlákny a vlákny jádra existuje několik návazností – modelů, které jsou zobrazeny na obrázku 4.6.



Obrázek 4.6: Možné návaznosti uživatelských vláken a vláken jádra.

- Model „**mnoho k jednomu**“ přidružuje více uživatelským vláknům jeden proces jádra. Jádro uživatelská vlákna nespravuje a jejich existenci nerozlišuje. Práce s vlákny je zajištěna pomocí speciální **vláknové knihovny** (thread library), která pracuje v uživatelském režimu. Knihovna provádí vytváření, plánování, přepínání a rušení vláken. Nevýhodou je, že pokud vlákno zavolá blokující systémové volání, je blokován celý proces, tedy i ostatní vlákna téhož procesu. Jádro přepíná procesy, ne vlákna. Důsledkem je, že více vláken téhož procesu nemůže pracovat současně na systémech s více procesory (ovšem vlákna mohou být ve stavu čekání na událost). Další nevýhodou je nutnost dodatečné správy vláken, protože není zajišťována jádrem.
- Model „**jeden k jednomu**“ přidružuje uživatelským vláknům vlákna jádra – každé uživatelské vlákno má své korespondující vlákno v jádře. Správu vláken provádí jádro. Výhodou je, že další vlákna uživatelského procesu lze vykonávat i pokud nějaké volá blokující systémové volání – paralelní běh na více procesorech. Nevýhodou je, že pro každé uživatelské vlákno je vytvořeno vlákno v jádře. To přináší režii, která se negativně projevuje na výkonnosti systému.
- Model „**mnoho k mnoha**“ je kombinací předešlých modelů a přidružuje uživatelským vláknům stejný nebo menší počet vláken v jádře. Počet vláken v jádře může být omezen vzhledem k aplikaci nebo vybavení zařízení.

[7] Jádro může být také realizováno pomocí vláken.

4.4 Stavy činnosti procesů

Nový proces je vytvořen stávajícím procesem pomocí systémového volání. Proces, který založil další proces, je „**rodič**“ (parent) či „předek“. Vytvořený proces je „**potomek**“ (child). Vazba potomků na rodiče vytváří stromovou strukturu[8]. Pokud proces vytvoří další proces, jsou dvě možnosti pro proces rodiče: je dále vykonáván nebo se zablokuje a čeká na ukončení potomka.

Systémové volání (*fork*) vytvoří kopii volajícího procesu (rodiče), takže poté existují dva procesy se sdílenými částmi uživatelského kontextu. Toto systémové volání je voláno rodičem jednou, ale výsledek volání je vrácen dvakrát: do procesu rodiče a do procesu potomka. Volání vrátí PID nového procesu do rodiče a hodnotu $PID = 0$ do potomka. Dalším systémovým voláním (*execve*) se spustí programový kód potomka.

Procesy jsou svázány pomocí identifikátoru PPID (Parent PID), kterým potomek identifikuje rodiče. Ve výpisu 4.3[9] je zobrazen seznam procesů tvořící vrchol stromové hierarchie. Povšimněte si sloupců označených PID a PPID. První řádek zobrazuje informace o hlavním procesu, který vzniká automaticky při startu systému a je kořenem stromové struktury procesů. Jeho PID je rovno 1 a PID předka je rovno 0. Ve výpisu následují procesy o jednu úroveň níže ve stromové struktuře s různými PID, ale stejným PPID.

```
[]$ ps ax
UID  PID  PPID  CMD
0    1    0     systemd
0    2    1     migration
0    3    1     ksoftirqd
0    4    1     watchdog
0    5    1     migration
0    6    1     ksoftirqd
```

Výpis kódu 4.3: Proces *systemd* a jeho potomci.

Proces je ukončen, jak dokončí svůj kód a vyvolá systémové volání pro své odstranění (uvolnění zdrojů – paměť, otevřené soubory atd.). V některých OS není dovoleno, aby proces existoval bez rodiče. Pokud je proces rodič ukončen, jsou automaticky také ukončeni i jeho potomci nebo jsou převedeni pod jiného rodiče (například hlavní proces). Ukončení procesu může způsobit i jiný proces.

Proces se sám ukončuje pomocí systémového volání, například *exit*. Potomek může vrátit do rodiče informaci o jeho ukončení. Pro rozeznání, který potomek se ukončil, je rodiči předáno PID ukončeného procesu. Potomek je ve stavu „zombie“ mezi provedením systémového volání pro ukončení a předáním informace rodiči.

Změna rodiče procesu je zobrazena ve výpisu 4.4. Výpis zobrazuje situaci při spuštění skriptu `hierarchy.sh`, který pouze provede `ping 127.0.0.1`, tedy proces spustí další proces. Výpis zobrazuje, že proces *ping* je potomkem `hierarchy.sh`.

```
[]$ pstree -a | grep -B2 ping
```

[8] Neplatí pro všechny operační systémy.

[9] Výpis je zkrácen. Další zobrazené informace jsou ID uživatele (UID). $UID = 0$ označuje uživatele root. Poslední sloupec zobrazuje názvy procesů.

```
2  |   |-bash
3  |   |   `-sh hierarchy.sh
4  |   |       `-ping 127.0.0.1
```

Výpis kódu 4.4: Hierarchie procesů.

Při ukončení skriptu `hierarchy.sh` je potomek převeden pod hlavní proces *systemd* s $PID = 1$, jak je dokázáno ve výpisu 4.5. Naopak, pokud se ukončí řídící terminál, ve kterém byl spuštěn skript `hierarchy.sh` (TTY pst/0; zde se tiskne výstup příkazu `ping`), tak je ukončen i proces *ping*.

```
1 []$ ps j | egrep 'PID|ping'
2 PID   PPID  TTY    STAT TIME COMMAND
3 15694 1     pts/0  S    0:00 ping 127.0.0.1
```

Výpis kódu 4.5: Zdědění procesu.

Při popisu činností procesů bylo řečeno, že proces se může nacházet v několika stavech. Předešle popsaný základní stavový model je zjednodušená podoba komplexního modelu. Typický **devítistavový model běhu procesů** je znázorněn na obrázku 4.7 [27]. Jedná se o stavový diagram[10] popisující přechody mezi jednotlivými stavy procesu.

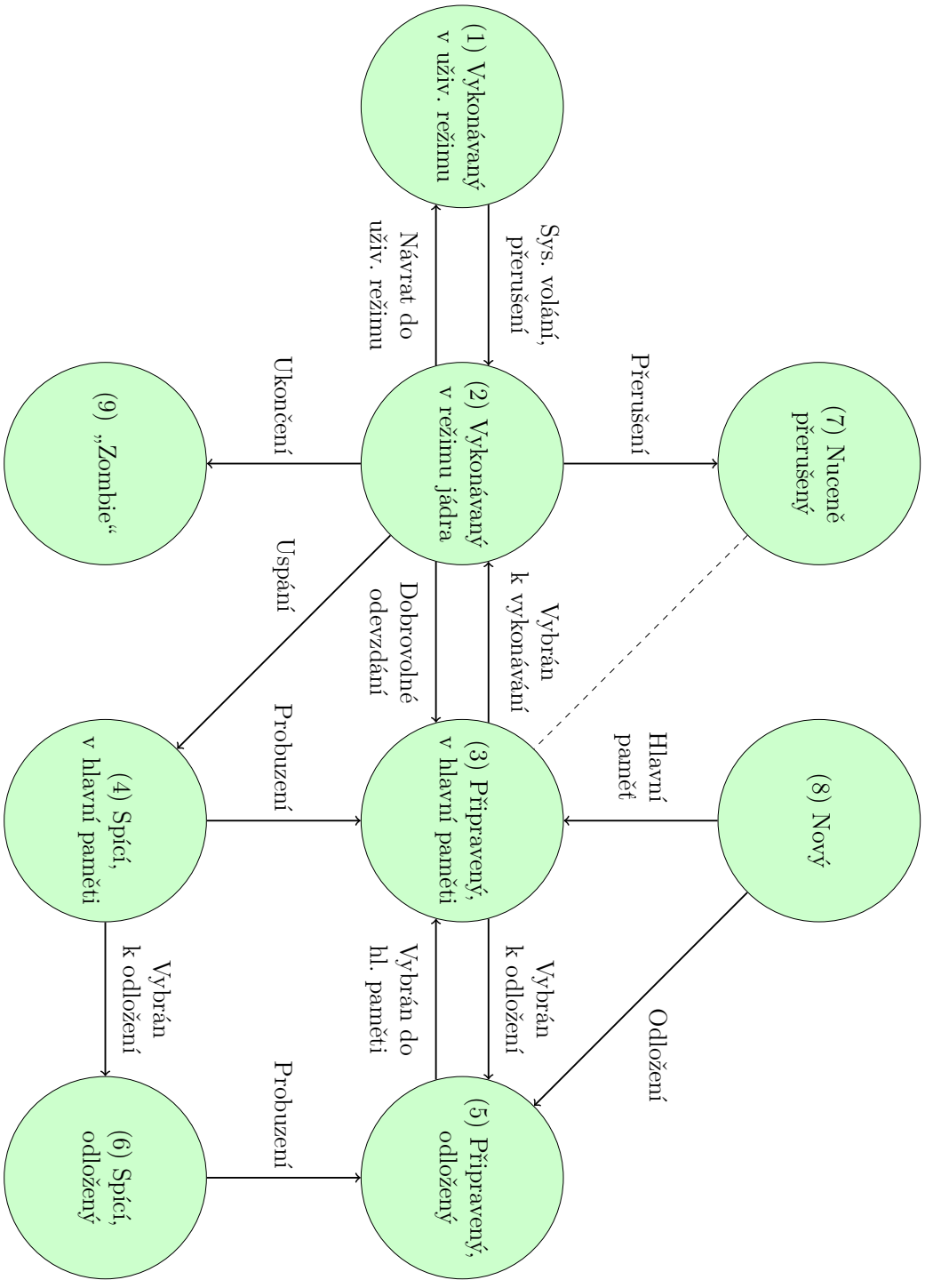
Stavy rozšířeného modelu jsou:

1. Proces vykonává vlastní kód
2. Proces vyvolal systémové volání nebo je prováděna systémová činnost (přerušení)
3. Proces je připraven ke zpracování a je uložen v hlavní paměti
4. Proces je spící a je uložen v hlavní paměti
5. Proces je připraven ke zpracování a je uložen v odkládací paměti
6. Proces je spící a je uložen v odkládací paměti
7. Vykonávání procesu je nuceně přerušeno
8. Proces je vytvořený a je v přechodovém stavu
9. Proces je ve stavu „zombie“, neexistuje, ale v systému je o něm záznam

Při návaznosti základního modelu na rozšířený lze pozorovat tyto stejné roviny činnosti.

- Rovina „vykonávání“ základního modelu je popsána dvěma stavy – „vykonávaný v uživatelském režimu“ a „vykonávaný v režimu jádra“. Přechod mezi těmito stavy je dán voláním systémového volání (které volá vykonávaný proces), nebo výskytem přerušení.
- Rovina „čekání“ základního modelu je popsána dvěma stavy – „spící a uložen v hlavní paměti“ a „spící a uložen v odkládací paměti“. Přechod mezi těmito stavy je dán plánovačem II. úrovně, který vyměňuje procesy mezi hlavní a odkládací pamětí. Zde dochází pouze k přechodu směrem do odkládací paměti.

[10] Orientovaný graf, jehož uzly představují stavy, do kterých může proces vstupovat, a hrany reprezentují události, které způsobují přechod procesu z jednoho stavu do druhého.



Obrázek 4.7: Rozšířený stavový model běhu procesu.

- Rovina „připravený“ základního modelu je popsána dvěma stavy – „připraven ke zpracování a uložen v hlavní paměti“ a „připraven ke zpracování a uložen v odkládací paměti“. Přechod mezi těmito stavy je opět dán plánovačem II. úrovně, který vyměňuje procesy mezi hlavní a odkládací pamětí. Zde dochází k obousměrným přechodům.

Dále rozšířený model popisuje roviny z pohledu umístění procesu. Jsou to:

- Rovina „proces v hlavní paměti“ – zahrnuje stavy „připraven ke zpracování a uložen v hlavní paměti“ a „spící a uložen v hlavní paměti“. Přechod mezi těmito stavy je jednosměrný a je způsoben probuzením procesu, tj. dočkáním se na událost, kvůli které byl proces uspán.
- Rovina „proces v odkládací paměti“ – zahrnuje stavy „připraven ke zpracování a uložen v odkládací paměti“ a „spící a uložen v odkládací paměti“. Přechod mezi těmito stavy je stejný jako u předešlého případu.

Pro jednoduchost prozatím budeme při popisu stavového modelu uvažovat, že procesy jsou v paměti vždy celé. Pohled na procesy z pohledu jejich umístění v paměti bude později rozšířen o možnost dělení procesu na části. Některé části budou v hlavní paměti a jiné zase v paměti odkládací, viz kapitola 5.

Vytvoření procesu

Prvním z možných stavů procesu je stav „vytvořený“. Do dalšího stavu přejde proces podle aktuální velikosti volného místa v hlavní paměti. V obou následujících stavech je proces připravený ke zpracování, rozdíl je v tom, ve které paměti se proces nachází. Předpokládejme, že proces přejde do stavu „připraven ke zpracování v hlavní paměti“. Plánovač I. úrovně[11] časem proces vybere k vykonávání a proces přejde do stavu „vykonávaný v režimu jádra“, kde se dokončí volání jádra, které obslouží jeho vytvoření. Připomeňme si, že systémové volání je obslouženo v režimu jádra, proto je dokončeno až ve stavu „vykonávaný v režimu jádra“. Po ukončení práce v režimu jádra přejde proces do uživatelského režimu, kde začne vykonávat svůj programový kód – stav „vykonáván v uživatelském režimu“.

Vykonávání procesu

Vykonávání procesu je popsáno dvěma stavy, podle toho v jakém režimu je proces vykonáván – uživatelský režim a režim jádra. Proces přechází mezi režimem jádra a uživatelským režimem podle volání systémových volání (volá vykonávaný proces) a obsluhy přerušení. Po zpracování systémového volání dojde k návratu do uživatelského režimu vykonávání procesu.

Při preemptivním střídání procesů má každý proces povoleno vykonávání do určité maximální doby (viz kapitola 4.5). Pokud tato maximální doba vykonávání uběhne, je generováno přerušení. Dále nedojde k návratu do uživatelského režimu, ale proces přejde do stavu „nuceně přerušený“. Plánovač vybere jiný proces k vykonávání. Pro lepší pochopení tohoto stavu ho lze přirovnat k nucenému blokování. Ovšem rozlišujme mezi tímto stavem, a stavem „spící“, kdy proces čeká na nějakou událost, kterou sám požaduje. Stav „nuceně přerušený“ je ve skutečnosti totožný se stavem „připraven ke zpracování v hlavní paměti“. Přerušovaná čára na obrázku, která spojuje tyto dva stavy, naznačuje jejich ekvivalenci. Rozlišením těchto stavů je zdůrazněna skutečnost, jak se proces do stavu „připraven ke zpracování v hlavní paměti“ dostal. Druhou možností je, že proces dobrovolně ukončí své vykonávání před uplynutím maximálního časového intervalu, který měl

[11] Plánování procesů bude rozebráno později v kapitole 4.5.

k dispozici (přechod označený jako „Dobrovolné ukončení“) a přejde do stavu „připraven ke zpracování v hlavní paměti“.

Uspání procesu

Předpokládejme, že proces vyžaduje vstupně/výstupní operaci, při které musí čekat na její dokončení[12]. Proces tedy zavolá systémové volání pro zápis/čtení dat a opustí stav „vykonávaný v uživatelském režimu“ a přejde do stavu „vykonávaný v režimu jádra“. V tomto případě by nebylo vhodné, aby byl proces stále v tomto stavu – docházelo by k plýtvání zdrojů výpočetního systému. Proto proces zablokuje sám sebe – uspí se – až do chvíle, kdy je informován o dokončení vstupně/výstupní operace. Proces přechází do stavu „spící v hlavní paměti“. Tímto je umožněno vykonávání dalšího procesu z množiny procesů ve stavu „připraven ke zpracování v hlavní paměti“. Když je později vstupně/výstupní operace dokončena, je proces probuzen a přechází do stavu „připraven ke zpracování v hlavní paměti“. Zde čeká, až plánovač proces vybere ke zpracování a tím dokončí svou vstupně/výstupní operaci.

Odložení procesu

Předpokládejme, že v systému je spuštěno více procesů a všechny tyto procesy se nevejdou do hlavní paměti. Některé z procesů musí být odloženy do odkládací (vedlejší) paměti. Přesuny procesů mezi hlavní a odkládací pamětí, tedy stavy „připraven v hlavní paměti“ a „připravený, odložený“ řídí plánovač II. úrovně, tzv. vyměňovací proces. Tento plánovač provádí výměnu procesů tak, aby každý z procesů byl někdy ve stavu „připraven v hlavní paměti“ a tím mu bylo umožněno, aby jej plánovač I. úrovně vybral k vykonávání (tento plánovač vybírá pouze z procesů, které jsou ve stavu „připravený v hlavní paměti“).

Stejná situace může nastat, pokud je proces ve stavu „spící v hlavní paměti“. V tomto případě plánovač II. úrovně přesune proces do stavu „spící, odložený“. Po probuzení (dočkání se na událost, která jej zablokovala) proces přechází do stavu „připravený, odložený“. Po určité době je proces vrácen do hlavní paměti – stav „připraven v hlavní paměti“ tak, aby mohl být vybrán plánovačem I. úrovně k vykonávání.

Ukončení procesu

Pro ukončení své činnosti provede proces systémového volání ze stavu „pracující v uživatelském režimu“. Systémové volání se vykoná ve stavu „pracující v režimu jádra“ a dále následuje stav „zombie“. V tomto stavu proces neexistuje, nicméně jsou o něm stále udržovány informace v systému. Jakmile rodič procesu zpracuje informaci o ukončení svého potomka, tak je proces definitivně ukončen.

[12] Zde se nemusí jednat pouze o vstupně/výstupní operace. Dalším příkladem je uspání procesu do doby, než nastane nějaká specifikovaná událost. Proces může být například uspán na dobu danou časovačem, jak bude uvedeno v příkladu na stav procesu „zombie“, viz kapitola 4.9.2.

4.5 Plánování procesů

Na počítači s jedním procesorem může být vykonáván jen jeden proces a ostatní musí čekat, až na ně přijde řada. Dochází k přepínání vykonávání procesů neboli ke změně kontextu. Ke změně kontextu dochází, když je více procesů ve stavu „připraven ke zpracování v hlavní paměti“ (viz obrázek 4.7). V tomto případě musí dojít k rozhodnutí, který proces bude vybrán. Tuto situaci řeší **plánování procesů** (scheduling). V operačním systému se o plánování procesů stará proces zvaný **plánovač** (process scheduler). Plánovač rozhoduje podle zvoleného **plánovacího algoritmu**.

Úkolem plánování procesů je efektivní přidělení času procesoru jednotlivým procesům. Správným plánováním můžeme přispět k lepší „činnosti“ systému. Dříve bylo úkolem rozdělit několik procesů na jeden procesor. Nyní plánování pracuje s rozdělováním více procesů na více procesorů.

Co však dělá situaci tak komplikovanou, že nelze použít jednoduché „spravedlivé“ střídání procesů? I tato možnost by jistě fungovala, ale představme si situaci, že používáme interaktivní systém a v něm existují dvě soupeřící úlohy. První aktualizuje obrazovku po uzavření okna a druhá odesílá email. Pokud bude zavření okna trvat např. 2 sekundy čekajíc, než se odešle email, dojem uživatele ze systému nebude dobrý. Když se okno uzavře okamžitě a email se odešle až následně, tak si této skutečnosti uživatel pravděpodobně nepovšimne. Závěr je, že při konfiguraci plánovače je třeba brát v úvahu několik faktorů, které ovlivňují činnost systému.

4.5.1 Okamžiky rozhodnutí

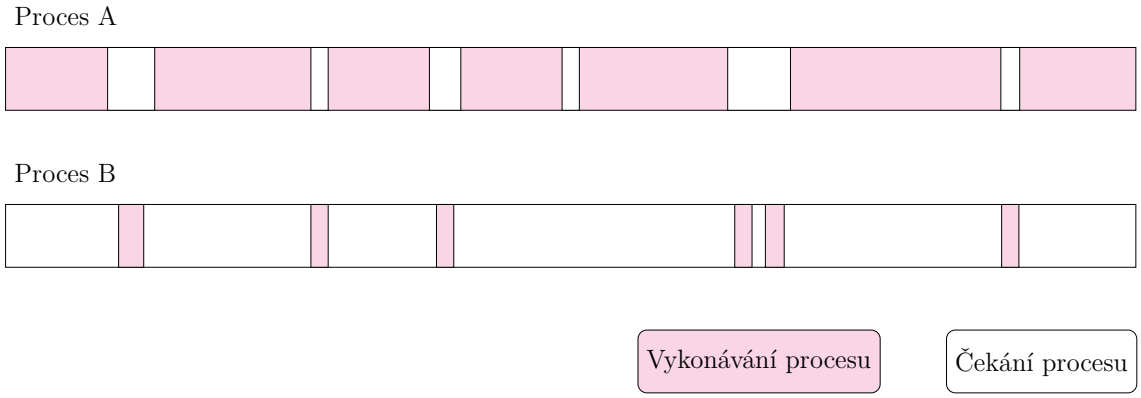
Existuje několik situací, kdy plánovač musí rozhodnout o tom, který proces bude vykonáván. Tyto situace se nazývají **okamžiky rozhodnutí** a jsou následující:

1. Vytvoření procesu – je nutné určit, zda bude vykonáván rodičovský nebo nově vytvořený proces.
2. Ukončení procesu – je nutné vybrat jiný proces k vykonávání.
3. Blokování procesu – například čekání na I/O operaci.
4. Přerušení procesu – a) přerušení od zařízení, b) přerušení od hodin.

Procesy střídají výpočetní a čekací úseky. Pokud proces čeká na I/O operaci, je blokován a jiný proces je vybrán k vykonávání. Časový průběh dvou procesů je vidět na obrázku 4.8. Některé procesy jsou více výpočetně zaměřené, zatímco jiné provádějí hodně I/O operací. V současné době je růst rychlosti CPU rychlejší (paralelní zpracování) než rychlost paměťových médií. Procesy tedy většinově čekají (na obrázku proces B).

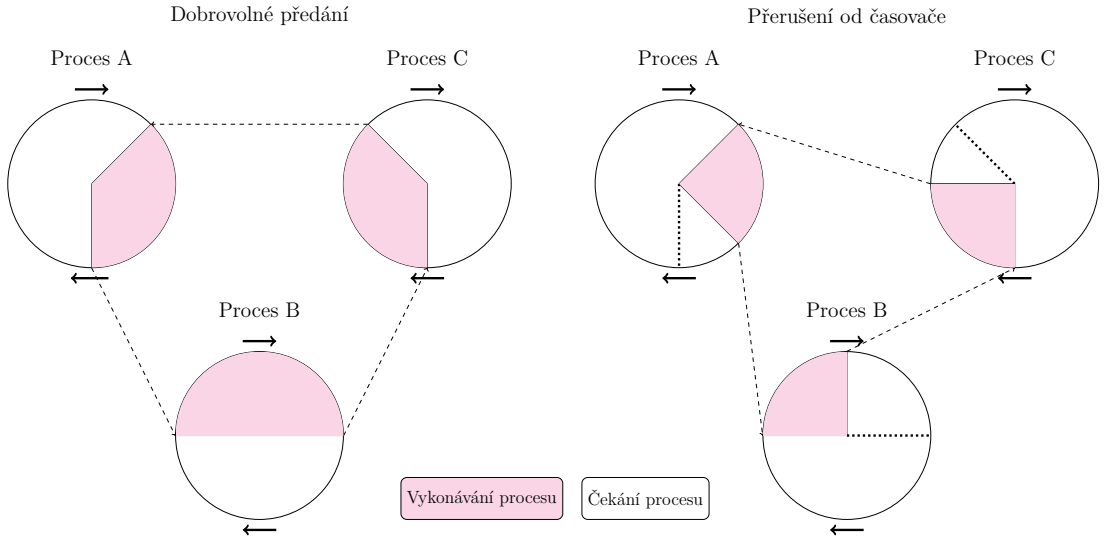
Rozhodnutí o přepnutí procesu může nastat při přerušení od zařízení (je potřeba obsloužit zařízení). Přepnutí procesu může být také na základě přerušení od hodin, např. při každém n-tém přerušení. Plánovač reaguje na přerušení od hodin podle toho, zda je systém **nepreemptivní**[13] (kooperativní), či **preemptivní**, viz obrázek 4.9. V prvním

[13] Z anglického slova preemption, což znamená nucenou výměnu.



Obrázek 4.8: Střídání výpočetního intervalu s intervalem čekání pro a) proces s dlouhými intervaly výpočtů b) proces s krátkými intervaly výpočtů a převažujícími I/O operacemi.

případě nedochází k přepínání procesu, dokud vykonávaný proces neuvolní procesor dobrovolně (není tedy závislé na přerušení od hodin). Ve druhém případě je při přerušení od hodin vykonávání procesu **nuceně přerušeno** a plánovač zvolí jiný proces.



Obrázek 4.9: Kooperativní a preemptivní přepínání procesů.

4.5.2 Typy systémů

Počítačové systémy slouží různým účelů a plánovací algoritmus tuto skutečnost zohledňuje. Rozlišují se tři základní typy systémů: **dávkový** (batch), **interaktivní** a **reálného času** [11, 16].

Dávkové systémy

V dávkových systémech jsou úlohy zpracovávány v **dávkách**. Dávky jsou zpracovávány bez účasti uživatele – nejsou zde prodlevy způsobené čekáním na akce od uživatele. Typicky se používá nepreemptivní (kooperativní) přepínání procesů. Tento typ systému se

používá ve výkonných výpočetních centrech. V dávkových systémech jsou sledovány tyto parametry:

- Výkonnost – počet úloh zpracovaných za jednotku času, například 100 úloh zpracovaných za hodinu.
- Rychlost obsluhy – průměrný čas od přijetí požadavku do jeho vyřízení.
- Využití CPU – preferuje se maximální využití a minimální prostoje.

Plánovací algoritmus, který zvyšuje výkonnost, nemusí zvyšovat rychlost obsluhy. Pokud jsou preferovány časově kratší úlohy, výkonnost roste (počet zpracovaných úloh). Časově delší úlohy však čekají na své zpracování a klesá tak rychlost obsluhy.

Příkladem dávkových operačních systémů pro výpočetní centra jsou z/OS od firmy IBM a GCOS (General Comprehensive Operating System) od firmy Bull SAS.

Interaktivní systémy

U interaktivních systémů jsou procesy řízeny akcemi uživatele (interakce). Tyto požadavky uživatelů by měly být vyřizovány přednostně. Interaktivní systémy jsou tedy v kontrastu oproti dávkovým systémům. Například otevření nebo uzavření okna by mělo být provedeno okamžitě na úkor stahování dat ze sítě. Interaktivní operační systémy se používají u osobních počítačů. Plánovač nemá vliv na složitost úlohy a rychlost jejího zpracování. Může však přispět ke splnění očekávání uživatele. V těchto systémech se používá preemptivní přepínání, aby se znemožnilo zabrání CPU procesem na delší dobu.

V interaktivních systémech se sleduje:

- Zpoždění odezvy – čas od zadání příkazu do obdržení výsledku.
- Přiměřenost – vztah složitosti úlohy k době jejího zpracování.

Uživatelé mají určitou představu (správnou či nesprávnou) o složitosti požadované úlohy. Pokud složitá úloha (vnímána jako složitá) trvá delší dobu, uživatel to přijme více příznivěji, než když čeká dlouhou dobu na zpracování jednoduché úlohy (vnímané jako jednoduché). Například uvažme navazování spojení v síti vs. jeho ukončení. Uživatel je mnohem ochotnější čekat na začátek relace, než na její ukončení.

Příkladem interaktivních systémů jsou běžné systémy jako Windows, MacOS, Linux.

Systémy reálného času

Systémy reálného času se používají pro zpracování úloh, které slouží jako vstup navazujících systémů. Tyto systémy musí (by měly) dokončit určitou úlohu v daném čase a reagovat na vnější události průběžně, tedy v **reálném čase**. Real-time operační systémy jsou používány například v robotice a telekomunikacích. Používá se zde preemptivní plánování procesů.

Rozlišujeme dva systémy podle zajištění požadavku na splnění úkolu ve stanoveném čase:

- Měkké (soft) – časové záruky jsou přibližné; jsou možné určité časové odchylky v reakcích.

- Tvrdé (hard) – časové záruky jsou plně zajištěny.

U tvrdých systémů může být vyžadováno použití pouze statické alokace paměti. U dynamické alokace se udržují seznamy volných bloků paměti, jejich prohledání a zabrání přináší časové zdržení. Dále systémy reálného času mohou mít předpoklad „nekonečné práce“, tedy bez restartu či dalšího zásahu. Použití dynamické paměti může způsobit, že alokované části paměti nebudou řádně uvolněny (chyba v aplikaci, která při statické alokaci není možná). Tato situace může vést po delší době ke kolapsu systému. U systémů reálného času se také nepoužívá koncept virtuální paměti, viz kapitola 5.

U real-time systémů jsou sledovány tyto parametry:

- Dodržování časových termínů – nejdůležitější požadavek.
- Předvídatelnost/pravidelnost – například u systémů pro zpracování multimediálních dat.

Například v multimediálních systémech pro zpracování zvuku není ztráta dat kritická. Výpadky lze dopočítat či nahradit předchozími daty. Ovšem pokud dochází k nepředvídatelnému přepínání procesu, který má na starosti zvukový výstup, tak může být snížena celková kvalita zvuku (nepravidelné přehrávání).

Příkladem systémů reálného času jsou FreeRTOS (open source) a VxWorks od firmy Wind River.

Na základě znalosti času potřebného pro obsloužení periodických událostí můžeme prohlásit, zda je real-time systém schopný činnosti. Uvažme m periodických událostí. Událost i se vyskytuje s periodou P_i a vyžaduje C_i času procesoru pro své zpracování. Real-time systém musí vyhovět následující podmínce [11]:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1. \tag{4.1}$$

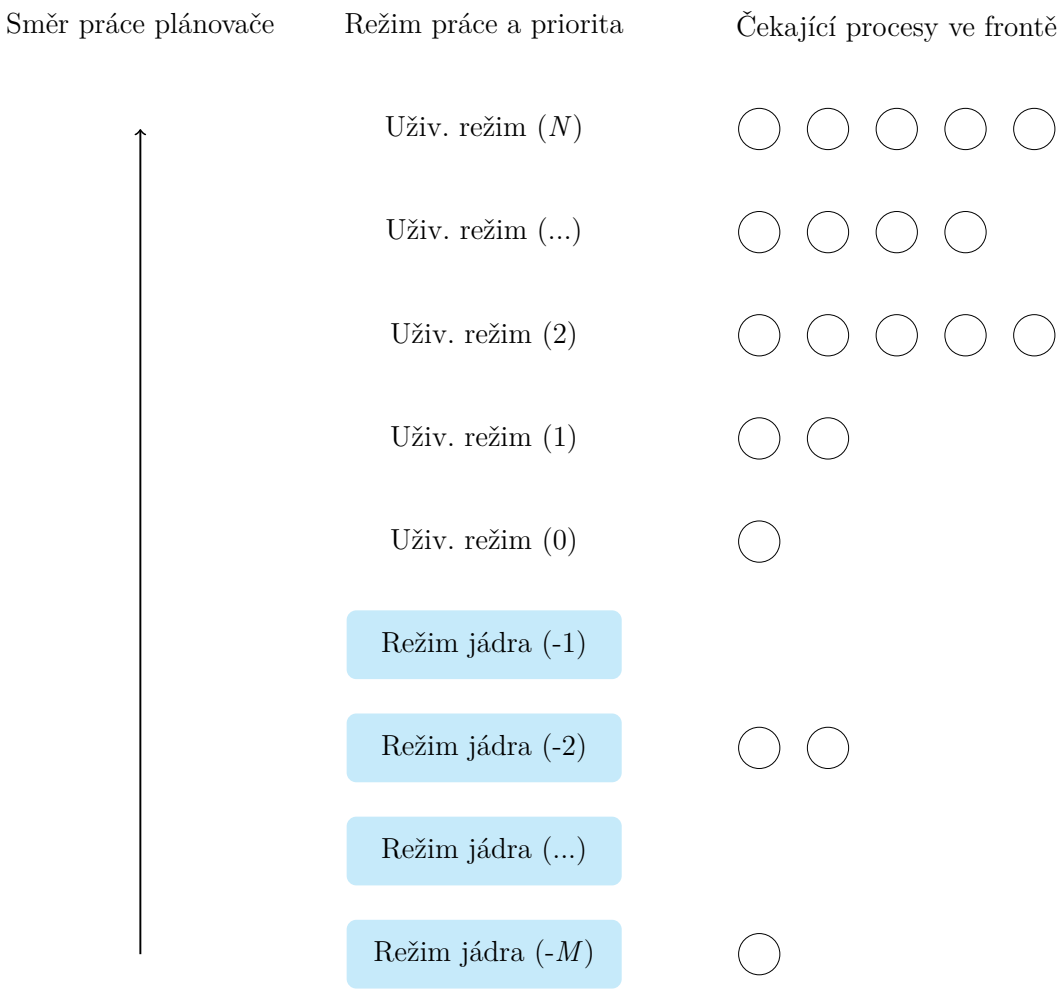
Jako příklad uvažme situaci se třemi periodickými událostmi $m = 3$ s těmito periodami výskytu $P_1 = 100$, $P_2 = 200$ a $P_3 = 500$ ms. Tyto události vyžadují pro svou obsluhu $C_1 = 50$, $C_2 = 30$ a $C_3 = 100$ ms. Systém vyhovuje, protože $0,5 + 0,15 + 0,2 < 1$.

4.5.3 Činnost plánování

Plánování procesů může být realizováno několika způsoby. Obecné plánování [11, 27] pracuje ve dvou úrovních:

- I. úroveň vybírá procesy, které jsou ve stavu činnosti „připraven ke zpracování v hlavní paměti“ tak, aby mohly přejít do stavu „vykonávaný v režimu jádra“ či „vykonávaný v uživatelském režimu“, viz obrázek 4.7.
- II. úroveň přesunuje procesy mezi hlavní a odkládací pamětí tak, aby všechny procesy mohly být vybrány plánovačem I. úrovně (bude vysvětleno dále v kapitole 5 zabývající se správou paměti). Ve vztahu ke stavu činnosti procesů na obrázku 4.7 se jedná o přesuny mezi stavy „připraven ke zpracování v hlavní paměti“ a „připravený, odloženy“. V případě spícího procesu je zapojen i stav „spící, odložený“.

Obecný plánovací algoritmus I. úrovně je postaven na **paralelních frontách**, každá z nich je spojena s intervalem nepřekrývajících se **priorit**, viz obrázek 4.10. Procesy v uživatelském režimu mají priority kladné. Naopak procesy v režimu jádra (tj. provádí se systémové volání, přerušení) mají priority záporné. Negativní hodnoty značí vyšší prioritu a pozitivní hodnoty značí nižší prioritu. Toto pojení se může lišit napříč operačními systémy, například u Windows je tomu obráceně. V plánovacích frontách jsou pouze procesy, které jsou ve stavu „připravený ke zpracování v hlavní paměti“.



Obrázek 4.10: Plánovač s paralelními frontami procesů.

Plánovač prohledává fronty od nejvyššího intervalu priorit (to jest u nejzápornějších hodnot). Až nalezne frontu, která je obsazena, vybere první proces z této fronty. Proces je vykonáván a poté vložen zpět do své fronty. S procesy v rámci jedné fronty je zacházeno cyklicky (round-robin).

Pro každý proces je priorita P pro počátek zvoleného intervalu (například každou sekundu) přepočítána podle vzorce [11]:

$$P = V + N + B \tag{4.2}$$

Podle nové priority je proces zařazen do příslušné fronty. Příslušná fronta je typicky zvolena na základě dělení vypočtené priority konstantou.

- Hodnota V představuje využití procesoru. V je zvyšováno o jedna každým taktem hodin, při kterém je proces je vykonáván. Zvyšováním hodnoty V se proces posunuje do fronty s nižší prioritou. Tímto způsobem jsou starší (vykonávané) procesy penalizovány a upřednostňovány jsou novější procesy. Operační systémy mohou používat různé způsoby penalizace starších procesů. Například předešlá hodnota V je navýšena o počet taktů z aktuálního intervalu a výsledek je podělen dvěma. Takto je poslední přidaná hodnota taktů hodin dělena dvěma, předchozí přidaná hodnota je dělena 4 a tak dále. Tímto způsobem nedávné využívání procesoru penalizuje proces více než využívání procesoru před nějakou dobou.
- Každý proces má přiřazenu hodnotu N (nice[14]). Tato hodnota je implicitně nastavena na nulu, ale povolený rozsah je od -20 do $+19$ (typicky). Běžný uživatel může hodnotu nastavovat v rozmezí $0 - 19$. Tím může „penalizovat“ své procesy. Administrátor má oprávnění „preferovat“ všechny procesy tím, že sníží hodnotu nice v rozmezí -20 až -1. Hodnotu N procesy dědí od svého rodiče.
- Hodnota B (báze) je použita, pokud se proces zablokuje (čeká na I/O operaci). Proces je odstraněn ze své plánovací fronty. Až nastane událost, na kterou proces čekal, je vložen do plánovací fronty, ale s jinou prioritou. Volba fronty závisí na události, na kterou proces čekal – hodnota B).

Příklad konkrétních hodnot priority pro procesy je zobrazen ve výpisu 4.6. Sloupeček *PR* značí prioritu, *NI* značí hodnotu nice.

```
[]$ top
PID USER   PR   NI   %CPU   %MEM COMMAND
43  root   39   19   0,0    0,0  khugepaged
42  root   25   5    0,0    0,0  ksmd
926 rtkit  21   1    0,0    0,0  rtkit-daemon
1   root   20   0    0,0    0,1  systemd
```

Výpis kódu 4.6: Hodnoty priority procesů.

4.6 Komunikace mezi procesy

Procesy mohou pracovat nezávisle nebo mohou spolupracovat. Nezávislý proces neovlivňuje jiné procesy a ani není ovlivňován. Každý proces, který si nevyměňuje data s jiným procesem, je procesem nezávislým. **Spolupracující** proces je ovlivňován nebo ovlivňuje jiné procesy.

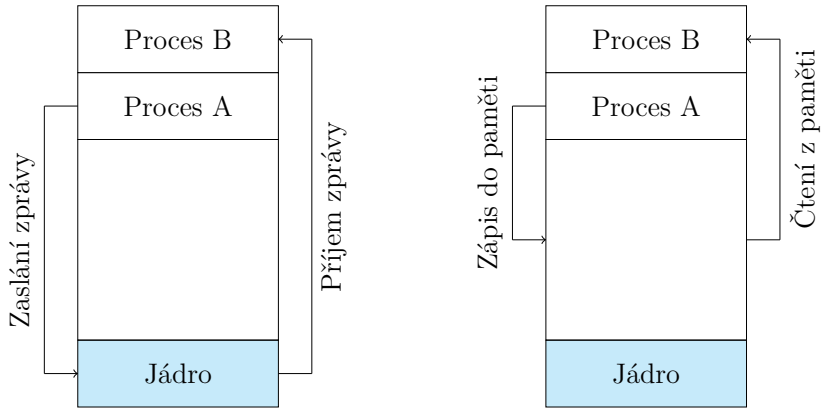
Výměna dat mezi procesy je prováděna pomocí **sdílené paměti** a **zasílání zpráv** [11, 16].

- Sdílená paměť – v paměťovém prostoru je vyhrazena oblast, do které mají přístup oba komunikující procesy. Procesy si vyměňují informace čtením a zápisem z/do této sdílené paměti.

[14] Překládá se nejčastěji jako „ohleduplnost“.

- Zasílání zpráv – informace jsou vyměňovány pomocí zaslání bloku dat. Zprávy mohou být také posílány přes systémová volání prostřednictvím jádra.

Způsoby obou mechanismů komunikace jsou uvedeny na obrázku 4.11. Proces A předává informaci procesu B. U předávání zpráv je zobrazen případ komunikace pomocí jádra. V případě sdílené paměti je informace procesem A zapsána na místo ve sdílené paměti a proces B si ji následně přečte.



Obrázek 4.11: Způsoby komunikace mezi procesy.

Zasílání zpráv

Zaslání zpráv je vhodné pro přenos menšího objemu dat. Nedochází zde ke konfliktům. Také je jednodušší na implementaci než výměna informací pomocí sdílené paměti. Odesílání i přijímání zpráv může pracovat ve dvou režimech:

- Blokující režim – Vysílací proces je blokován, dokud není zpráva přijata druhým procesem. Přijímací proces je blokován, dokud není zpráva dostupná.
- Neblokující režim – Vysílací proces dál pokračuje ve své činnosti, aniž by čekal na doručení zprávy. Přijímací proces testuje, zda je zpráva dostupná – pokud ne, tak pokračuje dál ve své činnosti.

Jednoduchým příkladem zasílání zpráv je propojení výstupu jednoho procesu na vstup druhého procesu pomocí **roury** (pipe). Je zde použit blokující režim – proces, který chce číst data je blokován, dokud nejsou data k dispozici (dokud nepřijme zprávu, že může data číst). Roura se zapisuje jako `cmd1 | cmd2`, kde

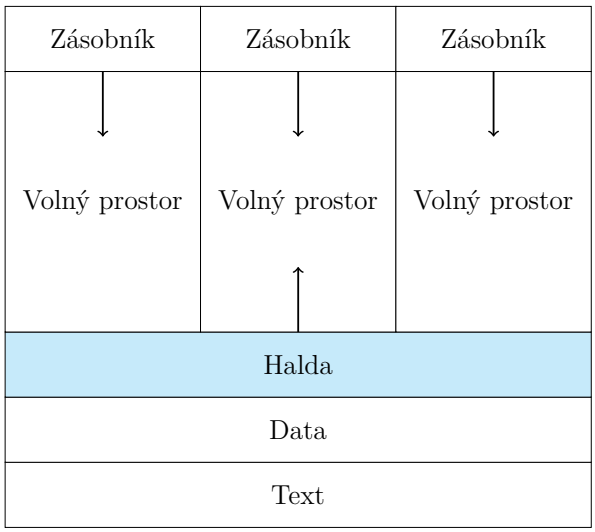
- vstup příkazu `cmd1` je napojen na klávesnici,
- výstup příkazu `cmd1` je předáván na vstup příkazu `cmd2`,
- výstup příkazu `cmd2` je propojen s obrazovkou.

Příkladem je výpis obsahu adresáře po stránkách `ls | more`. Je vytvořena dočasná paměť, do které první příkaz zapisuje. Jak je přijímacímu procesu oznámen konec zápisu – zaslána zpráva, tak tento proces začne z paměti číst. Po ukončení práce obou procesů je dočasná paměť uvolněna. Další možností je komunikace pomocí zápisu do souboru, tzv. pojmenovaná roura. Pro použití této roury je potřeba předávací soubor vytvořit. Tento soubor není automaticky smazán po ukončení komunikace. Jednoduchý test práce

pojmenované roury je vytvoření předávacího souboru *roura* pomocí příkazu `mkfifo roura` a následné zobrazení jeho obsahu `cat roura`. V druhém terminálu lze zapisovat text do roury pomocí přesměrování klávesnice do souboru `cat > roura`. V prvním terminálu bude po ukončení zápisu do roury (znak nového řádku) – zaslání zprávy, tento text zobrazen.

Sdílená paměť

Sdílená paměť je oproti zasílání zpráv vhodná pro přenos velkého objemu dat. Procesy si stejný blok paměti přičlení do uživatelského kontextu. Pokud do paměti zapíše jeden proces, budou tato data ihned viditelná ostatním procesům, které přistupují ke stejné sdílené paměti. Dalším způsobem sdílení paměťového prostoru pro čtení/zápis je i společná dynamická paměť pro vlákna jednoho procesu – halda, viz obrázek 4.12 a popis realizace vláken v kapitole 4.3.



Obrázek 4.12: Sdílení paměti pro čtení/zápis mezi vlákny jednoho procesu.

Použití sdílené paměti může způsobit **konflikty**. Konflikt nastává, když proces čte ze sdílené paměti dříve, než jiný proces dokončí zápis (stav souběhu) – jsou načteny nekonzistentní data. Tato situace může nastat, když vykonávaný proces zapisuje data do sdílené paměti a je přepnut kontext – začne se vykonávat jiný proces. Nově vykonávaný proces může číst data ze sdílené paměti (předchozí proces přitom nedokončil zápis). Proto je potřeba práci těchto procesů **synchronizovat**.

4.7 Synchronizace procesů

Ve víceúlohových systémech je spuštěno současně více procesů (vláken), které mezi sebou komunikují. Proces může ovlivnit jiný proces, nebo být ovlivněn jiným procesem. V této kapitole se zaměříme na zajištění synchronizace procesů. Seznámíme se i s klasickým problémem synchronizace.

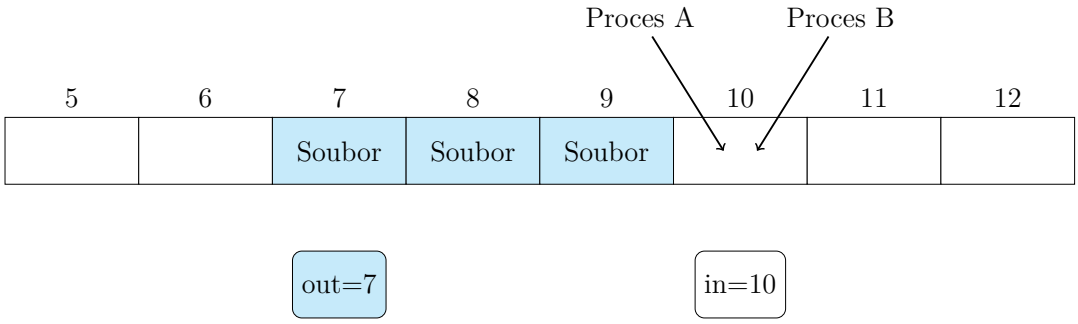
4.7.1 Požadavky synchronizace

Základní požadavky na synchronizaci procesů jsou [11, 16]:

- vyloučení konfliktu kritických činností pro zamezení souběhu,
- zajištění časových mezníků pro paralelní programování.

Konflikt kritických činností (souběh)

Proč potřebujeme vyloučit konflikt kritických činností? Důvod je práce procesů se sdílenými prostředky. Představme si situaci sdílené tiskové fronty. Proces, který chce tisknout, uloží soubor do tiskové fronty. Tiskový démon pravidelně kontroluje frontu a dokument vytiskne. Fronta se skládá z několika míst (slotů) a pracuje v režimu FIFO (první vložený soubor jde první na tisk). K frontě jsou definovány dva ukazatele: *out* a *in*. Ukazatel *out* značí místo ve frontě, ve kterém je uložený dokument pro vytištění v dalším kroku. Ukazatel *in* značí místo, do kterého proces uloží soubor k tisku. Schéma fronty je vyznačeno na obrázku 4.13.



Obrázek 4.13: Příklad práce tiskové fronty.

Předpokládejme, že ve frontě je několik souborů pro tisk. Tiskový démon vytiskl tři soubory a ukazatel *out* je nastaven na 4. Ukazatel *in*, ukazující na první volné místo, má hodnotu 10. Dva procesy se „rozhodnou“ téměř ve stejném okamžiku uložit do fronty svůj soubor pro tisk. Proces A si přečte ukazatel *in* a hodnotu 10 uloží do své lokální proměnné, kterou nazvěme *volny_slot*. Plánovač rozhodne, že bude vykonáván proces B (dojde k přepnutí kontextu). Proces B si také přečte ukazatel *in* a stejně tak uloží hodnotu 10 do své lokální proměnné *volny_slot*. V této chvíli oba procesy uvažují jako další volné místo č. 10. Proces B uloží soubor pro tisk do místa 10 a zvýší hodnotu ukazatele *in* o jedna na hodnotu 11. Poté proces B ukončí svou činnost. Plánovač vybere proces A a ten pokračuje v ukládání svého souboru. Ve své lokální proměnné *volny_slot* má uloženu hodnotu 10 a zapíše tedy soubor do místa 10, čímž přepíše soubor procesu B. Hodnotu ukazatele *in* nastaví proces A na 11. Tiskový démon vytiskne soubory z fronty. Jeden z uživatelů však marně stojí před tiskárnou a čeká na svůj dokument.

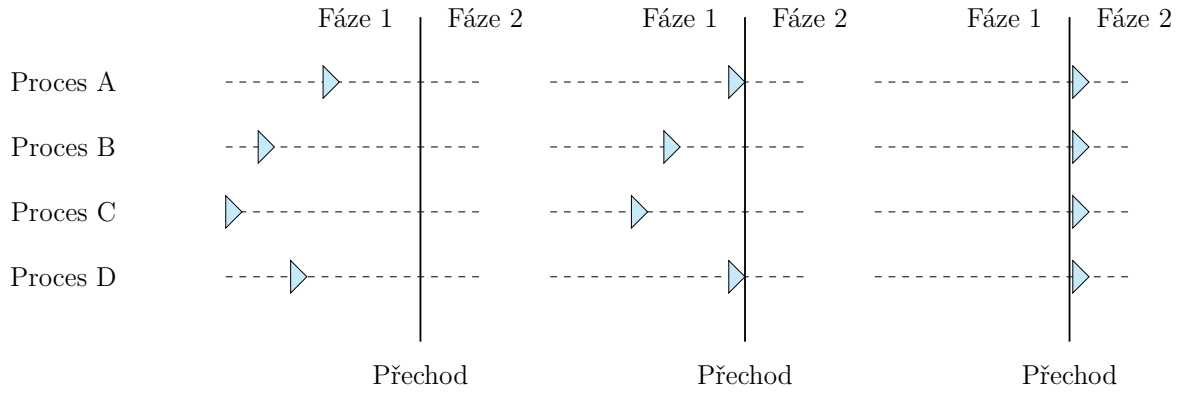
Podobné situace, kdy dva a více procesů přistupují ke sdíleným prostředkům a výsledek závisí na pořadí a čase přístupu, se nazývá **souběh** (race condition). Pokud vykonávaný

proces zapisuje data do sdílené paměti a je přepnut kontext (začne se vykonávat jiný proces), druhý proces může číst **nekonzistentní data** z oblasti, do které zapisoval předchozí proces a ještě nedokončil svoji práci.

K souběhu nemusí dojít a program pracuje správně. Ovšem dle Murphyho zákonů program v době jeho testování pracuje správně, ale když je spuštěn v reálné aplikaci[15], vyskytne se problém. Z principu této chyby je obtížné ji odhalit. Je proto třeba již při vývoji dbát na ošetření možného souběhu.

Časové mezníky

Zajištění časových mezníků se používá v případech, kdy proces musí čekat na dokončení množiny úloh, než může dále pokračovat. Typicky se jedná o paralelní programování pomocí více vláken jednoho procesu nebo více procesů. Pro tento účel lze použít **časové mezníky** (barriers), které označují společné setkání. Vlákna jednoho procesu mohou pracovat různými rychlostmi, i když vykonávají stejný kód. Příklad aplikace realizované pomocí vláken je zobrazen na obrázku 4.14. Vlákna musí dokončit svůj výpočet, než mohou přejít do další fáze. Tuto situaci si lze představit při zpracování obrazových dat velkého formátu, kdy vlákna jednoho procesu zpracovávají části obrazu a na konci každé fáze musí být vytvořena podoba celého obrazu pro nadcházející fázi.



Obrázek 4.14: Časová souslednost procesů.

4.7.2 Vzájemné vyloučení a kritická sekce

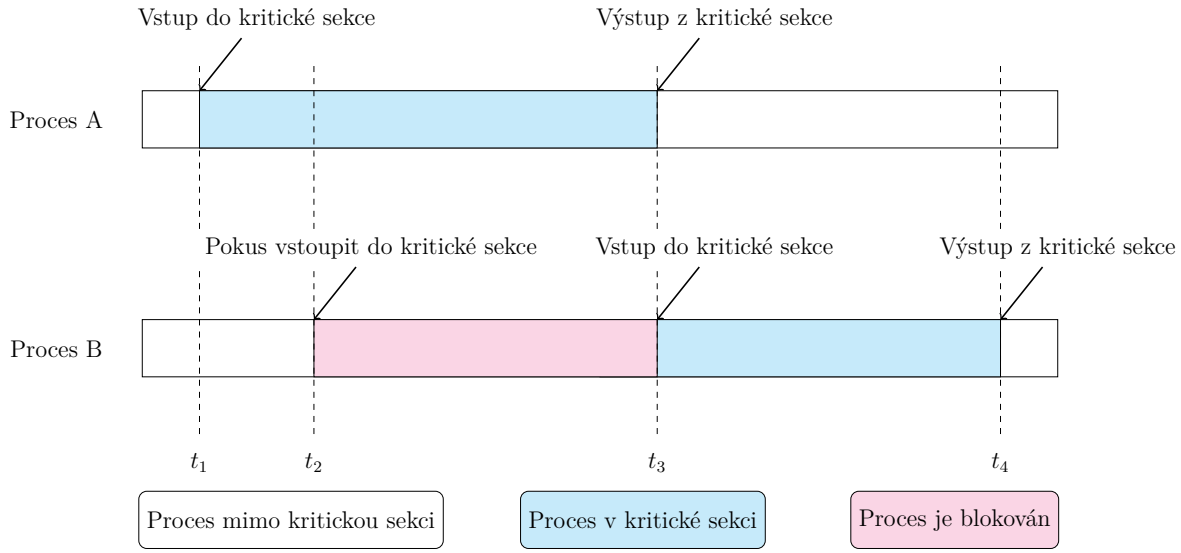
V následujících kapitolách se budeme zabývat prvním a náročnějším požadavkem synchronizace a to zamezením souběhu. Připomeňme si, že situace, kdy dva nebo více procesů přistupují ke sdíleným prostředkům a výsledek závisí na tom, v jakém pořadí a v jakém časovém okamžiku k nim přistoupí, se nazývá souběh.

Jak lze zabránit souběhu? Pomocí vyloučení čtení a zápisu sdílených dat ve stejném okamžiku – tzv. **vzájemné vyloučení** (mutual exclusion). Ke sdílenému prostředku musí být zajištěn výhradní přístup pro jeden proces po celou dobu jeho práce s tímto prostředkem. Synchronizace procesů spočívá v tom, že pokud jiný proces žádá o využití stejného prostředku, přejde do stavu čekání, dokud tento prostředek není uvolněn.

[15]Nebo při obhajobě BP/DP.

Zajištění vzájemného vyloučení řešíme tak, že ve zdrojovém kódu programu vymezíme část kódu, která přistupuje ke sdíleným prostředkům. Tato část kódu se nazývá **kritická sekce** (také kritická oblast). Principem je zabránit dvěma a více procesům ve stejném čase vstoupit do svých kritických sekcí. Pokud se dosáhne výlučného vstupu do kritických sekcí, je dosaženo zabránění souběhu.

Příklad dvou procesů s řízeným přístupem do svých kritických sekcí je na obrázku 4.15. Proces A vstoupí do své kritické sekce v čase t_1. Druhý proces B chce vstoupit do své kritické sekce v čase t_2, který je menší než čas t_3 (čas opuštění kritické sekce procesem A). Tento pokus ale selže, neboť v kritické sekci je již jiný proces (proces A). Proces B je tedy pozastaven do času t_3, kdy proces A opouští svou kritickou sekci. V té chvíli je procesu B umožněno vstoupit do své kritické sekce. V čase t_4 proces B opouští svou kritickou sekci.



Obrázek 4.15: Kritické sekce procesů.

Pro zajištění vzájemného vyloučení jsou ve zdrojovém kódu zařazeny dvě části, viz výpis 4.7:

- Část starající se o vstup do kritické sekce.
- Část starající se o výstup z kritické sekce.

```
do{
 kod procesu
 vstupni sekce
 kriticka sekce
 vystupni sekce
 kod procesu
}while(TRUE);
```

Výpis kódu 4.7: Vstupní a výstupní sekce pro zajištění vzájemného vyloučení.

Obecná myšlenka vzájemného vyloučení je postavena na principu **zámku** nad sdíleným prostředkem. Proces před vstupem do kritické sekce zámek zkontroluje, a pokud je

„otevřený“, zámek „zamkne“ a vstoupí. Po výstupu z kritické sekce zámek „uvolní“ a další proces může vstoupit do své kritické sekce. Při popisu možných řešení se nejdříve zaměříme na intuitivní přístupy a vysvětlíme si, v čem spočívá jejich problematické použití. Postupně se dostaneme k řešením, která se používají.

Zajištění vzájemného vyloučení dělíme na softwarové a hardwarové. Základními způsoby jsou [11, 16]:

- softwarové, pomocí proměnných,
- hardwarové, pomocí přerušení a atomických instrukcí,
- softwarové, pomocí systémových volání.

Mimo vzájemné vyloučení je nutné ještě zajistit další základní podmínky:

1. Blokování jiného procesu je možné pouze v případě, že proces je v kritické sekci. Není přijatelné, aby byl proces blokován jiným, který pracuje mimo svou kritickou sekci.
2. Doba čekání na vstup do kritické sekce musí být konečná.

4.7.3 Vzájemné vyloučení pomocí proměnných

Základní typy zajištění vzájemného vyloučení pomocí proměnných jsou [11, 16]:

- sdílená dvouhodnotová proměnná,
- striktní střídání procesů.

Sdílená dvouhodnotová proměnná

Sdílená proměnná *lock* (zámek nad sdíleným prostředkem) je počátečně nastavená na hodnotu FALSE. Při *lock* = FALSE není žádný proces ve své kritické sekci. Při *lock* = TRUE je jeden z procesů ve své kritické sekci.

Pokud chce nějaký proces vstoupit do své kritické sekce, zkontroluje proměnnou *lock*. Pokud přečte hodnotu FALSE, nastaví *lock* na TRUE a vstoupí. Jestliže přečte hodnotu TRUE, přejde do čekací smyčky, neboť je v kritické sekci jiný proces. V čekací smyčce probíhá neustálá kontrola proměnné *lock*.

Při tomto jednoduchém přístupu ovšem nastává stejný problém jako v případě tiskové fronty. Uvažujme situaci, že jeden z procesů přečte hodnotu *lock* = FALSE, ale než ji stihne nastavit na TRUE dojde k přepnutí kontextu. Jiný proces je vybrán pro vykonávání a provede také čtení proměnné *lock*. Tento proces zjistí stejnou hodnotu (FALSE) a nastaví TRUE. Poté vstoupí do své kritické sekce. Nyní je opět přepnut kontext a je vykonáván první proces. Původní proces pokračuje tam, kde byl přerušen, tedy nastaví TRUE a vstoupí také do své kritické sekce. Výsledkem jsou dva procesy v kritické sekci najednou.

Striktní střídání procesů

Další možností je, že procesy si vzájemně předávají možnost vstupu do kritické sekce – striktně se střídají. K tomu lze využít proměnnou, která určuje, který proces může

vstoupit do své kritické sekce. Zvolme si proměnnou *turn*, která může nabývat hodnot 0 a 1 pro dva procesy A a B. Prvně chce do své kritické sekce vstoupit proces A, čte $turn = 0$ a vstoupí. Pokud v tuto chvíli čte proměnnou proces B, zjistí že $turn = 0$. Proces B vstoupí do čekací smyčky, ve které probíhá neustálá kontrola proměnné *turn*. Když proces A dokončí svou kritickou sekci, přepne proměnnou *turn* do stavu 1. Proces B nyní může vstoupit do své kritické sekce. Po jejím ukončení přepne proces B proměnnou *turn* na 0. Nyní je řada na procesu A. Blíže viz výpis 4.8.

```
-- Proces A --
while (TRUE) {
 while (turn != 0) {}  /*cekej az na mne bude rada*/
 kriticka sekce
 turn = 1;
 ne kriticka sekce
}
-- Proces B --
while (TRUE) {
 while (turn != 1) {}  /*cekej az na mne bude rada*/
 kriticka sekce
 turn = 0;
 ne kriticka sekce
}
```

Výpis kódu 4.8: Striktní střídání procesů.

Problém tohoto řešení nastává, když některý z procesů vstupuje do své kritické sekce častěji než jiný proces – proces A žádá o vstup do kritické sekce předtím, než provedl kritickou sekci proces B. Pokud je $turn = 1$, je nyní řada na procesu B. Tím je porušena jedna z uvedených základních podmínek – žádný proces nesmí blokovat jiný, pokud není ve své kritické sekci.

4.7.4 Vzájemné vyloučení pomocí hardware

Zajištění vzájemného vyloučení pomocí hardware lze řešit [11]:

- zákazem přerušení,
- atomickými instrukcemi.

Zákaz přerušení

Připomeňme si, že v preemptivních systémech je proces vykonáván jen po určitou dobu a pak dojde k přerušení. Proces však může být přerušen i z jiného důvodu – např. obsluha vstupně/výstupního zařízení. Řešením může být zakázání všech přerušení, pokud je nějaký proces v kritické sekci. Pokud není přerušení povoleno, tak vykonávaný proces nemůže být přerušen a jiný proces nemůže vstoupit do své kritické sekce. Přerušení mohou zakazovat uživatelské a systémové programy. Umožnění uživatelským procesům zakazovat přerušení však není moudrý přístup, neboť takovéto procesy mohou způsobit pád systému. Uživatelské programy do operačního systému přidávají uživatelé a je jejich volbou, který program instalují. Pokud není program dobře napsán (nebo zlomyslně tuto

chybu obsahuje), může zablokovat všechna přerušení a pak je nikdy neobnovit. Možností je zákaz přerušení povolit pouze pro systémové procesy.

Přístup pomocí zakázání přerušení pracuje jen u jednoprocesorových systémů. Pokud je u víceprocesorových systémů na jednom procesoru zakázáno přerušení, tak tento zákaz není platný pro jiný proces vykonávaný na jiném procesoru. Při úvaze zakázání přerušení na všech procesorech je třeba posílat zprávy všem procesorům, což je časově neefektivní.

Atomické instrukce

Další možností je použití hardwarových instrukcí, které umožní najednou číst a měnit proměnnou nebo zaměnit obsah dvou proměnných. Tyto instrukce jsou označovány jako **atomické**, protože je nelze přerušit.

Princip si vysvětlíme na základní instrukci *TestAndSet*. Instrukce *TestAndSet* (označována také jako TSL – Test and Set Lock) testuje sdílenou proměnnou. Pokud je její hodnota rovna FALSE, je proměnná nastavena na TRUE, a proces vstoupí do kritické sekce. Pokud se jiný proces pokusí o vstup, tak čeká v aktivní smyčce. Po dokončení kritické sekce je sdílená proměnná opět nastavena na hodnotu FALSE. Princip je tedy podobný jako u sdílené proměnné. Rozdíl je v tom, že instrukce *TestAndSet* je nepřerušitelná, což zajišťuje vzájemné vyloučení (nemůže dojít k tomu, že proměnná bude přečtena, ale nebude nastavena).

4.7.5 Vzájemné vyloučení pomocí systémových volání

Všechny předchozí přístupy využívají aktivní čekání ve smyčce, když proces nemůže vstoupit do své kritické sekce. Tato smyčka může:

1. Plýtvat časem CPU – lze použít, pokud je předpoklad, že čekání nebude trvat dlouho.
2. Porušit základní podmínku, že doba čekání na vstup do kritické sekce musí být konečná.

Uvažme dva procesy s rozdílnými prioritami, proces H s vyšší prioritou a proces L s nižší prioritou. Plánování procesů podle priorit je postaveno na principu, že proces H je vykonán vždy, když je připraven. V situaci, kdy se proces L nachází v kritické sekci a proces H chce vstoupit do své kritické sekce, proces H otestuje podmínku vstupu. Protože vstup není povolen, tak přejde do aktivního čekání ve smyčce. Podle plánovací politiky je prioritnímu procesu věnován čas procesoru. Proces L tedy nikdy nedostane příležitost z kritické sekce vystoupit a proces H bude věčně čekat na opuštění kritické sekce procesem L. Tento stav je nazýván **problémem opačné priority** (inverse priority problem) [11].

Jak tedy zabránit procesu vstoupit do kritické sekce a přitom se vyhnout aktivnímu čekání? Řešením je komunikace mezi procesy. Proces přejde do režimu spánku a čeká na probuzení od jiného procesu. Ani tento přístup však není bezchybný. Předpokládejme, že máme dva procesy, kdy jeden připravuje data (producent) a druhý (spotřebitel) tato data zpracovává (producer-consumer problem nebo také bounded-buffer problem) [11]. Procesy pracují se sdílenou pamětí, která má omezenou velikost pro N zpráv. Producent

zprávy do paměti ukládá a spotřebitel je z paměti čte. Pokud producent chce do paměti uložit novou položku a paměť je plná, tak přejde do stavu spánku. Jak spotřebitel z paměti odebere položku, tak producenta probudí. Podobně, pokud chce spotřebitel odebrat položku z paměti a zjistí, že paměť je prázdná, uspí se. Producent budí spotřebitele poté, co položku do předešle prázdné paměti vloží. Proces producenta je zobrazen ve výpisu 4.9 a proces spotřebitele je zobrazen ve výpisu 4.10. Pro sledování počtu položek v paměti je použita proměnná *pocet*. Komunikace mezi procesy probíhá pomocí zpráv, které jsou zasílány přes systémová volání.

```
#define N 1000 /*pocet volnych mist v pameti*/
int pocet = 0; /*cislo polozky v pameti*/

void producent(void)
{
 int polozka;
 while (TRUE)
 {
  /*nekonecna smycka*/
  polozka = vytvor_polozku();
  if (pocet == N) sleep(); /*pamet je plna, jdi spat*/
  vloz_polozku(polozka); /*vlozi polozku do pameti*/
  pocet = pocet + 1;
  if (pocet == 1) wakeup(spotrebitel); /*pamet byla prazdna -> probud
   spotrebitele*/
 }
}
```

Výpis kódu 4.9: Producent.

```
void zakaznik(void)
{
 int polozka;
 while (TRUE)
 {
  if (pocet == 0) sleep() /*bez spat, nic neni v pameti*/
  polozka = odeber_polozku(); /*vyjmi polozku z pameti*/
  pocet = pocet - 1;
  if (pocet == N - 1) wakeup(producent) /*pamet byla plna -> probud
   producenta*/
  spotrebuj_polozku(polozka);
 }
}
```

Výpis kódu 4.10: Spotřebitel.

Opět si popišme problém tohoto řešení, který spočívá ve **fatálním souběhu** (fatal race condition) jelikož procesy producenta a zákazníka pracují se sdílenými prostředky bez kontroly přístupu. Předpokládejme, že paměť je prázdná a spotřebitel přečetl hodnotu proměnné *pocet* rovnající se 0. V tomto okamžiku bylo přerušeno vykonávání procesu spotřebitele a začal se vykonávat proces producenta. Proces spotřebitele se nestihl uspat. Producent vložil položku do paměti a zvýšil proměnnou *pocet* o jedna na $pocet = 1$. Producent předpokládal, že hodnota proměnné před zvýšením byla 0 a spotřebitel byl

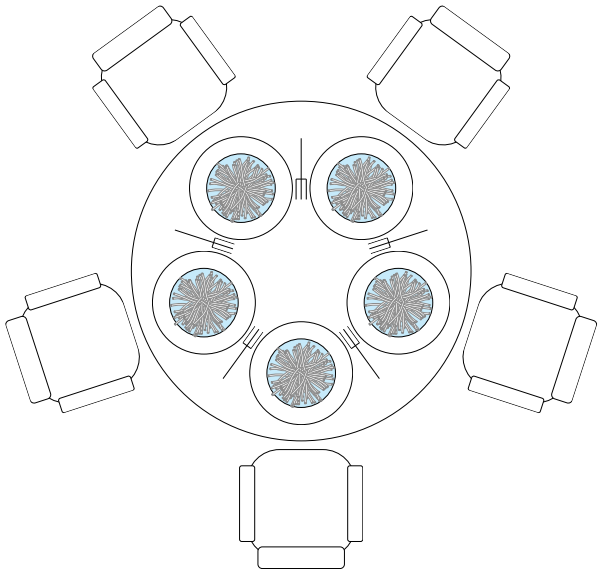
ve stavu spícím. Proto zavolá systémové volání *wakeup*, aby spotřebitele probudil. Avšak spotřebitel není ve stavu spícím, a tato zpráva o probuzení je ignorována (ztracena). Časem se začne vykonávat proces spotřebitele. Spotřebitel má již z minulého vykonávání přečtenou hodnotu proměnné $pocet = 0$ a další krok je, že se uspí. Producent zaplní paměť položkami a uspí se. Oba procesy dostanou do stavu spánku a spí jako Šípková Růženka; jsou ve fatálním souběhu.

Praktická řešení synchronizace jsou realizována pomocí semaforů a monitorů, viz příklad 4.9.4.

- Semafory poskytuje jako službu operační systém (realizováno jádrem). Jsou tedy nezávislé na programovacím jazyce.
- Synchronizace pomocí monitorů je řešena na úrovni programovacího jazyka. Monitory jsou abstraktní datové typy, které obsahují sdílené proměnné a funkce. Pokud chce proces přistupovat ke sdíleným prostředkům, musí tak učinit pomocí funkcí monitoru.

4.7.6 Klasický problém

V oblasti zkoumání synchronizace procesů byly publikovány zajímavé problémy. V polovině 60. let minulého století Dijkstra představil a navrhl řešení synchronizačního problému, který nazval „Dining philosopher problem“ [11]. Jeho zadání je jednoduché. Kolem kulatého stolu sedí pět filosofů a před každým je talíř špaget. Špagety jsou kluzké, k jejich konzumaci jsou třeba dvě vidličky. Na stole je pouze pět vidliček, které jsou umístěny mezi filozofy. Tento stůl je zobrazen na obrázku 4.16.



Obrázek 4.16: Stůl s večeří pro pět filosofů.

Filosofové při své večeři střídají jídlo a přemýšlení. Když filosof vyhladoví, pokusí se uchopit vidličky ležící po stranách jeho talíře a pokud se mu to podaří, začne jíst. Když se rozhodne přemýšlet, vidličky odloží na stůl. Špagety na talířích nikdy nedojdou a filosofové mohou jíst nekonečně dlouho. Úkolem programátora je napsat program tak, aby

se filosofové vždy najedli, tj. aby byli schopni donekonečna střídat fáze jídla a filozofování.

Intuitivním řešením je pro filozofa otestovat dostupnost levé vidličky, a když je volná, uchopit ji. Poté stejný postup opakovat u pravé vidličky. Když filosof dojí, odloží pravou vidličku a následně levou. Toto jednoduché řešení ovšem není správné. V málo pravděpodobné situaci (ale stále může nastat), kdy všichni filosofové najednou uchopí svou levou vidličku a budou chtít vzít pravou vidličku (která nebude dostupná), dojde k uvíznutí.

Tomu lze zabránit tím, že po uchopení levé vidličky si filozof zjišťuje dostupnost pravé vidličky po určitý časový interval. V případě, že pravá vidlička není do konce intervalu dostupná, filosof levou vidličku odloží, počká stejný interval a postup u levé vidličky opakuje. Toto řešení zabraňuje uvíznutí, ovšem časem dojde k tomu, že všichni filozofové ve stejný čas budou chtít jíst a uchopí levou vidličku, uvidí, že pravá není dostupná, čekají a pak odloží levou vidličku, čekají, uchopí levou vidličku, uvidí, že pravá není dostupná, čekají, odloží levou vidličku, čekají atd. Situace, kdy procesy pokračují nekonečně dlouho v opakování určité činnosti bez nějakého postupu, se nazývá **hladovění** (starvation) [30]. Tento pojem odvozen od problému hladovějících filosofů. Výskyt hladovění lze redukovat volením náhodných časů čekání mezi jednotlivými pokusy. Toto řešení je efektivní a v praxi často funguje tam, kde odložení provedení požadovaného úkonu není problém. Příkladem z oblasti síťové komunikace je kolize při přenosu technologií Ethernet – při kolizi stanice čekají náhodnou dobu, aby pokus zaslání dat zopakovaly. Ovšem v některých kritických aplikacích nelze provedení úkolu odkládat.

Pro aplikace, kde je nutné bezodkladně provést požadovanou akci, si představíme poslední řešení. Zde je zavedeno pořadí použití vidliček. Vidličky jsou očíslovány 1 až 5. Každý filosof se pokusí uchopit vidličku s nižším číslem a pak vidličku s vyšším číslem (samozřejmě myslíme dvě vidličky, které jsou u jeho talíře, „kradení“ vidliček je zakázáno). Pro odkládání vidliček není určeno pořadí. Jak toto řešení pracuje? V případě, že čtyři filozofové se zároveň pokusí uchopit vidličku s nižším číslem, jedna vidlička zůstane na stole – ta s nejvyšším číslem. Tedy pátý filosof nebude moci uchopit žádnou vidličku. Jiný z filosofů bude mít k dispozici vidličky dvě – jedna z nich bude ta s nejvyšším číslem. Nevýhoda tohoto řešení je, že se vždy nají pouze jeden filozof, ovšem mohli by dva.

4.8 Uvíznutí procesů

Ve víceúlohovém systému spolu soutěží množina procesů o množinu prostředků. Programy mohou navodit stav, kdy určitý proces využívá více prostředků a může tak nastat uvíznutí (deadlock). Skupina procesů je ve stavu uvíznutí, pokud každý z nich čeká na událost, kterou může způsobit pouze jiný proces z této skupiny. Uvíznutí může nastat v jednom systému nebo ve více systémech, pokud jsou prostředky dostupné po síti [30, 11, 16].

Příkladem mohou být dva procesy, kdy každý chce dokument uložený ve skeneru vypálit na CD. První proces zažádá o přístup ke skeneru a tento je mu přidělen. Druhý proces, který je naprogramován jiným způsobem, zažádá nejprve o CD-RW mechaniku a ta je mu přidělena. Nyní první proces požádá o CD-RW mechaniku, ale žádost je zamítnuta do té

doby, než druhý proces CD-RW uvolní. Druhý proces požádá o skener, ovšem se stejným výsledkem jako u prvního procesu. Procesy se tak dostaly do stavu uvíznutí.

Uvíznutí může nastat nad různými druhy prostředků, které mohou být hardwarové či softwarové. Tyto prostředky dělíme na nativně **nesdílitelné** (tiskárny, scannery, DVD) nebo **sdílitelné** (globální proměnná, sdílený úsek paměti, záznam v databázi). Sdílitelné prostředky mohou být uzamčeny pro použití pouze jedním procesem a být tak dočasně nesdílitelné. Příkladem může být uzamčení sdílitelného prostředku pro zajištění vzájemného vyloučení.

Prostředky jsou také děleny na **přepínatelné** (preemptable) a **nepřepínatelné** (nonpreemptable). Přepínatelný prostředek je takový, jehož odebrání procesu nezpůsobí žádnou škodu. Příkladem tohoto prostředku je hlavní paměť. Tato přidělená paměť může být procesu odebrána a data procesu přesunuta do odkládací paměti (viz princip virtuální paměti) beze ztráty informací. Na druhou stranu nepřepínatelný prostředek, je takový, jehož odebrání způsobí nefunkčnost procesu. Například pokud proces začne vypalovat CD, odebrání CD-RW mechaniky a její předání jinému procesu způsobí poškození dat.

Uvíznutí nenastává nad sdílitelnými prostředky bez jejich uzamčení či nad přepínatelnými prostředky, kde lze uvíznutí předejít předáním prostředků jinému procesu.

Při popisu uvíznutí budeme tedy uvažovat prostředky jako nesdílitelné, tj. může je využívat pouze jeden proces v jednom čase, a nepřepínatelné, tj. jejich odebrání procesu přináší ztrátu požadované funkce.

Využívání těchto prostředků se skládá ze tří kroků:

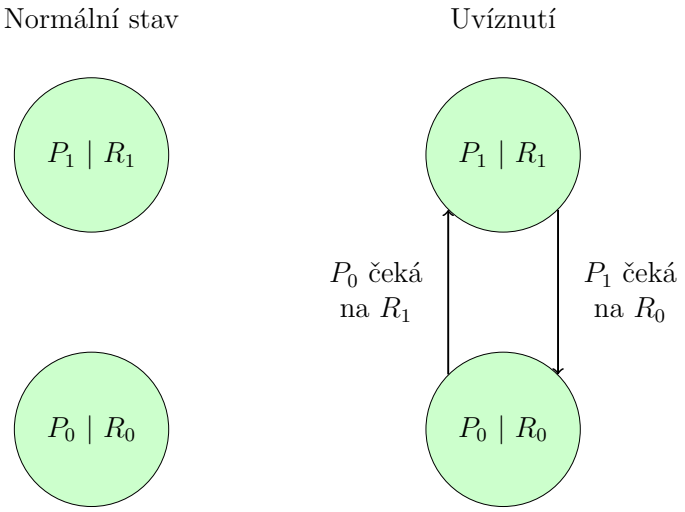
1. Žádost – Proces musí požádat o využití prostředku. Pokud je prostředek zrovna využíván, přejde proces do stavu čekání na uvolnění prostředku.
2. Používání – Proces využívá prostředek.
3. Uvolnění – Proces po ukončení práce předá zprávu o uvolnění prostředku.

Příčiny uvíznutí mohou být různé. Může se jednat o blokování procesů, kdy první proces čeká na prostředek obsazený druhým procesem, který je ve stavu čekání na prostředek obsazený prvním procesem, viz obrázek 4.17. K uvíznutí může také dojít při nekorektním chování (chyba v programu), například při ztrátě předávané informace o využívání prostředku.

Uvíznutí je (ne)řešeno v závislosti na typu operačního systému. Tři obecné přístupy jsou [16]:

1. předcházení uvíznutí,
2. připuštění, že uvíznutí může nastat; systém tento stav detekuje a obnoví normální činnost
3. ignorování problému uvíznutí.

Předcházení uvíznutí může být například řešeno zavedením číslování prostředků a vynucením pořadí při jejich používání od nižšího čísla k většímu (viz problém večeřících filozofů). Jako čísla prostředků mohou sloužit různé identifikátory, například unikátní adresa paměti svázaná s prostředkem.



Obrázek 4.17: Uvíznutí dvou procesů P0, P1 nad prostředky R0, R1. Normální stav – každý proces využívá jeden prostředek; uvíznutí – každý proces využívá svůj prostředek a čeká na prostředek využívaný druhým procesem.

Detekce uvíznutí a obnova normální činnosti vyžaduje pravidelné spouštění detekčního algoritmu. Tento algoritmus monitoruje využívání prostředků a stav procesů. Po zjištění stavu uvíznutí se používají dvě metody pro obnovení činnosti:

1. Procesy ve stavu uvíznutí jsou násilně ukončeny. Toto ukončení může proběhnout i) postupně – pomalejší řešení, po každém ukončení procesu je spuštěn detekční algoritmus znovu; volba procesu k ukončení může být například na základě stáří procesu, nebo ii) všechny najednou – rychlejší řešení s velkými náklady (ztráta předešlých výpočtů a dat).
2. Procesům ve stavu uvíznutí jsou násilně odebírány prostředky a předávány jiným procesům ve stavu uvíznutí. Po každém předání prostředku je spuštěn detekční algoritmus pro test, zda uvíznutí bylo vyřešeno

Ignorování problému uvíznutí se používá tam, kde je pravděpodobnost uvíznutí malá a jeho výskyt je tolerovaný. Výhodou je snížení režie operačního systému. Případné uvíznutí může řešit uživatel manuálním ukončením procesů.

4.9 Příklady na procesy

První příklad zobrazuje základní informace o procesech, jako je například čas jeho vzniku. Data procesů jsou uložena v jejich uživatelském kontextu. Tento kontext je rozebrán pomocí vzorového programu v druhém příkladu. Zde je demonstrována i změna rozložení uživatelského kontextu pomocí editace zdrojového kódu. Procesy se mohou nacházet v několika stavech. Zvláštním stavem je stav „zombie“. Pro navození tohoto stavu je použit stejný vzorový program. Tento program je spuštěn tak, aby stav „zombie“ byl vyvolán po určitou dobu a poté zanikl. Třetí příklad se zabývá komunikací mezi procesy pomocí zasílání zpráv – signálů. Využit je skript, který pracuje se signálem SIGINT, jehož účelem je proces přerušit. Dále je popsáno použití signálu SIGKILL pro zabití procesu. Poslední

čtvrtý příklad se zabývá synchronizací procesů. Dva procesy pracují se sdílenou proměnnou. Její hodnota je náhodná podle toho, kdy dojde k přepnutí vykonávání procesů. Tento problém je vyřešen pomocí semaforu.

4.9.1 Základní informace

Informace o procesech jsou k dispozici pomocí virtuálního souborového systému[16], který je dostupný v adresáři `/proc/`. Jedná se o formu rozhraní pro přístup k informacím o činnosti jádra. Každý proces svůj podadresář odpovídající jeho PID. Zde je k dispozici několik souborů, jejichž obsah zobrazuje stav činnosti daného procesu. Obsah těchto souborů je dynamicky generován na základě požadavku čtení. Například v souboru s názvem `cmdline` je zachycen název programu, kterým byl proces spuštěn, včetně předaných parametrů. Další základní informace zde dostupné jsou pracovní adresář, seznam otevřených souborů, rozložení paměti atd. Podle času vytvoření podadresáře lze zjistit času vytvoření procesu, jak je zobrazeno ve výpisu 4.11. Parametr `ls -d` zobrazuje adresáře jako soubory pro výpis času.

```
[]$ ls -ld /proc/1
dr-xr-xr-x. 9 root root 0 22. uno 17.49 /proc/1
```

Výpis kódu 4.11: Čas vytvoření procesu.

Stav činnosti procesu je dostupný v souboru `status`, jehož obsah je zobrazen ve výpisu 4.12. Zde je také uveden počet vláken procesu.

```
[]$ more /proc/1/status
Name:   systemd
State: S(sleeping)
Threads: 1
```

Výpis kódu 4.12: Stav jednoho procesu.

Přehled stavů více procesů lze získat příkazem `ps`, viz výpis 4.13. Význam vybraných stavů je: R – připraven ke zpracování, S – spí, T – zastaven a Z – zombie. Výpis také zobrazuje využití paměti a svázaný terminál.

```
[]$ ps u
USER   PID    %CPU  %MEM  VSZ   RSS   TTY    STAT COMMAND
bsosa  25263  0.0   0.1   5624  3360  pts/0  Ss   bash
bsosa  26330  0.0   0.0   2400  1020  pts/0  R+   ps u
```

Výpis kódu 4.13: Stav více procesů.

4.9.2 Uživatelský kontext a stav zombie

V uživatelském kontextu jsou uloženy informace, které proces potřebuje pro svou činnost. Jedná se o programový kód, statické proměnné a dynamicky alokovaná data. Základními

[16] Virtuální souborové systémy jsou popsány v kapitole 6.

částmi uživatelského kontextu jsou textová část, datová část, halda a zásobník. V následujícím příkladu se budeme zabývat statickými částmi uživatelského kontextu, tedy textovou a datovou částí. Jako vzorový program použijeme kód ve výpisu 4.14, který nazvěme `zombice`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  /*vytvoreni noveho procesu*/
  switch (fork())
  {
    /*potomek - vraceno 0*/
    case 0:  printf("Nejaka cool rodata zde\n");
             printf("PID %d\n", getpid());
             exit(0);
    /*rodic - vraceno PID potomka*/
    default: sleep(60);
             exit(0);
    }
}
```

Výpis kódu 4.14: Vzorový program pro uživatelský kontext a stav zombie.

Program po kompilaci (`gcc zombice.c -o zombice`) má uživatelský kontext zobrazený ve výpisu 4.15. Nyní může následovat otázka pozorného studenta, jak můžu pracovat s uživatelským kontextem, když program nebyl spuštěn? Odpověď je, že statické části (textová a datová) uživatelského kontextu jsou již přítomny ve zkompilovaném binárním souboru. Při spuštění programu se tyto části nakopírují do paměti a vytvoří se tak uživatelský kontext. Pro získání rozložení uživatelského kontextu byl využit program `size`, který zobrazuje jeho části ve zjednodušené podobě (viz dále).

```
[]$ size zombice
text   data   bss   dec    hex   filename
1653   588    4     2245   8c5   zombice
```

Výpis kódu 4.15: Obecné statické části uživatelského kontextu.

Zobrazené položky mají tento obsah:

- *text* – programové instrukce, textové řetězce, číselné hodnoty,
- *data* – inicializovaná data pro čtení a pro čtení i zápis,
- *bss* (Block Started by Symbol) – neinicializovaná data,
- velikost položek *text* a *data* v dekadickém a hexadecimálním zápisu.

Položka *text* také zahrnuje hodnoty pro inicializaci dat (textové řetězce, číselné hodnoty). Tyto hodnoty jsou při startu programu kopírovány do položky *data* uživ. kontextu – dojde tak k nastavení hodnot. Zobrazené obecné položky zahrnují další sekce dat, kterých je větší množství. Informace o bližším dělení uživatelského kontextu lze získat programem

`objdump`, který také využijeme k provedení experimentu. Pro přehlednost si zobrazme pouze vybrané části, jak je uvedeno ve výpisu 4.16. Tento výpis byl zjednodušen pro přehlednost.

```
[]$ objdump -h zombice | egrep "text|rodata|bss"
.text    01b2 READONLY, CODE
.rodata  002f READONLY, DATA
.bss     0004 ALLOC
```

Výpis kódu 4.16: Vybrané statické části uživatelského kontextu.

Ve výpisu je u každé části uvedena její velikost v hexadecimálním zápisu, vlastnosti (např. READ ONLY) a typ – programové instrukce nebo data (CODE, DATA). Povšimněte si, že zde část *.text* má menší velikost (0x1b2 = 434 B) než původní obecná položka *text* v předchozím výpisu (1653 B). Část *.text* již obsahuje pouze instrukce programu. Textové řetězce jsou uvedeny ve zvláštní části *.rodata* (47 B), kterou následně využijeme. Řádky s textovými řetězci programu `zombice` jsou zvýrazněny ve výpisu 4.17. Další části původní položky *text* jsou také již obsaženy jinde, pro náš účel je to však nepodstatné.

```
...
    /*potomek - vraceno 0*/
    case 0:  printf("Nejaka cool rodata zde\n");
             printf("PID %d\n", getpid());
             exit(0);
    /*rodic - vraceno PID potomka*/
...
```

Výpis kódu 4.17: Textové řetězce ve zdrojovém kódu programu.

Nyní prozkoumejme obsah části *.rodata* s textovými řetězci, jak je uvedeno ve výpisu 4.18. Na začátku výpisu je přidána informace o architektuře, pro kterou byl program zkompilován. Následuje zobrazení textových řetězců uvedených ve zkompilovaném programu `zombice`. Program `objdump` muže také být využit pro zobrazení zdrojového kódu v jazyce symbolických instrukcí. Tyto a další informace mohou být využity pro tzv. **reverzní inženýrství** (reverse engineering), kdy je potřeba „vytáhnout“ co nejvíce informací ze zkompilovaného programu. Například se tak může dát dopátrat toho, s jakými vstupními parametry program může být volán (pokud nejsou známy), či zda tam jsou nějaké skryté.

```
[]$ objdump -s -j .rodata zombice

zombice:     file format elf64-x86-64

Contents of section .rodata:
400780 ...  Nejaka cool roda
400790 ...  ta zde.PID %d..
```

Výpis kódu 4.18: Zobrazení sekce *.rodata* (textové řetězce) přeloženého binárního souboru.

Připomeňme si, proč dělíme data uživatelského kontextu na inicializovaná a neinicializovaná. Důvodem je zmenšení velikosti souboru s programem, který je menší o velikost neinicializovaných dat. Uveden je pouze počet bajtů pro alokaci při spuštění programu. Hodnota těchto dat může být náhodná nebo nastavena na nulu (ukazatele na „null“). Ne vždy je však výhodné použití neinicializovaných dat. Při programování mikrokontrolérů, kde může být velikost paměti RAM velmi omezená a dále může být relativní dostatek paměti FLASH, je naopak žádoucí program sestavit tak, aby sekce neinicializovaných měla co nejmenší velikost (tedy naopak zvětšit velikost souboru s programovým kódem). Část neinicializovaných dat *bss*, nebyla dosud ve zdrojovém kódu programu `zombice` zastoupena. Ve výpisu 4.15 je uvedena minimální velikost sekce (4 B). Jako další experiment provedeme to, že do zdrojového kódu přidáme tato neinicializovaná data, tak jak je zobrazeno ve výpisu 4.19. Následně budeme pozorovat změny v uživatelském kontextu a změny velikosti souboru.

```
long long a[1000]; /*velikost pro demostraci 8kB*/

int main(void)
{
  /*zjisteni velikosti long long*/
  printf("long long: %d\n", sizeof(long long));

  /*vytvoreni noveho procesu*/
  switch (fork())
   ...
```

Výpis kódu 4.19: Přidání neinicializovaných dat.

Do programu bylo přidáno pole o 1000 položkách proměnné typu *long long int*. Na 64bitové architektuře to odpovídá velikosti 8 kB. Povšimněte si, že deklarace byla uvedena globálně před funkcí *main*. Kdyby byla uvedena ve funkci *main*, tak by alokace proběhla v haldě. Program také nyní vypíše, jaká je velikost datového typu *long long*. Nyní program zkompilujeme a prozkoumáme, jak je uvedeno ve výpisu 4.20.

```
[]$ gcc zombice_bss.c -o zombice_bss
[]$ size zombice_bss
text   data   bss    dec    hex    filename
1684   588    8032   10304  2840   zombice_bss
```

Výpis kódu 4.20: Experiment s neinicializovanými daty.

Ve výpisu je vidět, že velikost položky uživatelského kontextu v přeloženém programu *bss* (neinicializovaná data) narostla o 8 kB ve srovnání se stavem ve výpisu 4.15 (hodnota nárůstu je přibližná z důvodu práce s bloky paměti).

Nyní prozkoumejme celkovou velikost souboru s programem (ne jenom uživatelského kontextu), viz výpis 4.21. Zde je vidět, že velikost souboru `zombice_bss` mírně narostla o přidané instrukce, ovšem nenarostla na dvojnásobnou velikost oproti původní hodnotě, tedy není navýšena o 8 KB.

```
[]$ls -l zombice
-rwxr-xr-x. 1 komosny komosny 8616 zombice
```

```
[]$ ls -l zombice_bss
-rwxr-xr-x. 1 komosny komosny 8640 zombice_bss
```

Výpis kódu 4.21: Porovnání velikosti souborů s přeloženým programem – původní a upravený.

Nové rozložení uživatelského kontextu pomocí programu `objdump` lze vidět ve výpisu 4.22. Zde je opět vidět nárůst sekce neinicializovaných dat (bss) na velikost 1f60h (8032), ve srovnání s výpisem 4.16. Připomeňme si, že neinicializovaná data nelze číst pomocí programu `objdump`, jelikož jsou dostupná až po spuštění programu.

```
[]$ objdump -h zombice_bss | egrep -A1 "text|rodata|bss"
.text   01c2 READONLY, CODE
.rodata 003e READONLY, DATA
.bss    1f60 ALLOC
```

Výpis kódu 4.22: Nové rozložení uživatelského kontextu dostupného v přeloženém binárním souboru po modifikaci programu.

Program `zombice` zužitkujme pro navození stavu „zombie“, což je obsahem vlastního zdrojového kódu. Stav vznikne tak, že proces rodič vytvoří proces potomek pomocí funkce *fork*. Proces rodič ihned přejde do stavu spící pomocí funkce *sleep*. Proces potomka nezavolá funkci *execve* pro spuštění jeho programového kódu, ale ihned zavolá funkci pro své ukončení *exit*. V programu se pouze před ukončením potomka vytiskne jeho PID. Rodič je zablokovaný (*sleep*) a nemůže tak dostat zprávu od potomka o jeho ukončení. Proces rodič má tedy stále informaci, že proces potomek existuje. Stav „zombie“ je závěrečným stavem existence procesu.

Program rozlišuje procesy rodiče a potomka podle návratového kódu funkce *fork*. Funkce vytvoří kopii volajícího procesu (rodiče) a v systému existují dva totožné procesy se sdílenými částmi uživatelského kontextu. Výsledek funkce je vrácen dvakrát: do procesu rodiče je vráceno PID potomka a do procesu potomka je vráceno $PID = 0$. Tyto stěžejní části jsou zvýrazněny ve výpisu 4.23.

```
...
  /*vytvoreni noveho procesu*/
  switch (fork())
  {
    /*potomek - vraceno 0*/
    case 0: printf("PID %d\n", getpid());
            exit(0);
    /*rodic - vraceno PID potomka*/
    default: sleep(60);
             exit(0);
...
```

Výpis kódu 4.23: Klíčové části programu pro stav zombie.

Ve výpisu 4.24 je pak zobrazen postup vyvolání stavu „zombie“. Program je spuštěn na pozadí pomocí znaku &, z důvodu možnosti další práce s terminálem. Nejprve je vytištěno PID procesu rodiče. Dále následuje PID vytvořeného potomka. Přehled vybraných procesů je zobrazen příkazem `ps u`. Zde jsou vyznačeny procesy rodiče a potomka s

korespondujícím PID. U procesu rodiče je zobrazen stav *S* – spí, protože je zablokován funkcí *sleep*. U procesu potomka je zobrazen stav *Z* – zombie. Také jeho název je upraven na `[zombice] <defunct>`. Po uplynutí času spánku dojde k probuzení rodiče, ten přijme zprávu o ukončení potomka a stav „zombie“ zanikne. Ve výpisu je také zobrazena jedna netradiční věc. Jedná se překročení rozsahu hodnot PID, kdy nový proces rodiče a potomka má menší číslo (666, 667), než procesy spuštěných terminálů (31922, 32687).

```
[]$ ./zombice &
[1] 666
PID 667
ps u
USER        PID  TTY    STAT START  COMMAND
komosny     666  pts/1  S    13:41  ./zombice
komosny     667  pts/1  Z    13:41  [zombice] <defunct>
komosny     672  pts/1 R+    13:41  ps u
komosny   31922  pts/1 Ss    13:06  bash
komosny   32687  pts/2 Ss+   13:31  bash
```

Výpis kódu 4.24: Stav procesu „zombie“.

4.9.3 Zasílání zpráv mezi procesy

Procesy mohou komunikovat pomocí zasílání zpráv. Přenos těchto zpráv může probíhat v blokujícím nebo neblokujícím režimu. Blokující režim značí, že proces čeká, dokud není zpráva přijata přijímacím procesem; přijímací proces čeká, dokud není zpráva dostupná. Zprávy v blokujícím režimu lze přenášet pomocí roury. Proces do roury zapíše blok dat. Následuje značka (zpráva), že přijímací proces může data číst. Roury mohou používat jako komunikační prostředek soubory, v tomto případě se jedná o pojmenované roury. Pojmenovanou rouru lze vytvořit pomocí příkazu `mkfifo`. Výpis 4.25 zobrazuje vytvoření roury a následné zobrazení jejího obsahu. Ve výpisu 4.26 je pak zobrazeno zadávání zpráv do roury, které probíhá v druhém terminálu. Po přenosu bloku dat (znak nového řádku) jsou data zobrazena v prvním terminálu.

```
[]$ mkfifo roura
[]$ cat roura
zprava druhy proces
dalsi zprava pro druhy proces
```

Výpis kódu 4.25: Předávání zpráv mezi procesy – zobrazení textu.

```
[]$ cat > roura
zprava druhy proces
dalsi zprava pro druhy proces
^C
```

Výpis kódu 4.26: Předávání zpráv mezi procesy – zadávání textu.

Zprávy lze také předávat prostřednictvím jádra, zde se jedná o tzv. signály. Signály jsou typicky posílány v neblokujícím režimu a může je odesílat uživatelský proces, systémový

proces nebo jádro. Signály mohou být generovány systémem, například při problémech práce s pamětí[17], nebo je může zasílat uživatel.

Uživatel může signály zasílat pomocí znaků v terminálu. Signál SIGTSTP zaslaný pomocí [Ctrl+z] pozastaví proces v daném terminálu a převede jej na pozadí. Seznam pozastavených procesů lze získat pomocí příkazu `jobs`. Pozastavený proces lze později obnovit a přenést na popředí, jak je ukázáno ve výpisu 4.27.

```
[]$ sh signal.sh
^Z
[1]+  Pozastavena    sh signal.sh
[]$ jobs
[1]+  Pozastavena    sh signal.sh
[]$ fg %1
sh signal.sh
```

Výpis kódu 4.27: Práce s procesy pomocí signálů – pozastavení a obnovení procesu.

Pomocí znaků [Ctrl+c] lze zaslat signál SIGINT, která nenásilně ukončí proces na popředí v daném terminálu. Tento signál lze zachytit a provést určité akce před ukončením procesu, viz výpis 4.28. Skript ve výpisu pracuje ve smyčce. Po příjmu signálu SIGINT je spuštěna funkce, která vypíše text a skript ukončí.

```
#!/bin/bash
trap funkce SIGINT

funkce()
{
 echo "Signal byl zachycen"
 exit
}

#nekonecna smycka
while true
do
 sleep 1
done
```

Výpis kódu 4.28: Zachycení signálu.

Oba signály SIGTSTP a SIGINT může proces zachytit a provést akce před jejich provedením. Některé signály však proces zachytit nemůže. Příklad rozdílu si uveďme pro dva signály zasílané příkazem `kill`:

- SIGTERM – nenásilně ukončí proces; signál může být zachycen; určeno pro korektní ukončení procesu.
- SIGKILL – násilně ukončí proces; signál nemůže být zachycen; určeno pro ukončení procesu v případě problémů.

[17] Například signál SIGSEGV, který je generován při odkazu na adresu mimo uživatelský kontext procesu.

Signál SIGTERM[18] pro nenásilné ukončení procesu je zaslán příkazem `kill`, kde parametrem je PID procesu, kterému se má signál zaslat (`kill 9374`). Signál SIGTERM je výchozím signálem příkazu `kill`.

Signál SIGKILL pro násilné ukončení procesu je opět zaslán příkazem `kill`, ovšem prvním parametrem je typ signálu (číslo 9 odpovídá signálu SIGKILL) a druhým parametrem je PID procesu pro ukončení (`kill -9 9245`).

Použití obou signálů pro ukončení procesu je uvedeno ve výpisu 4.29.

```
#nenasilne ukonceni - lze zachytit - signal SIGTERM
[]$ sh signal.sh &
[]$ ps a
PID  TTY      STAT   TIME COMMAND
9374 pts/0    S      0:00 sh signal.sh
[]$ kill 9374
[]$ ps a
...
[1]+  Ukoncen (SIGTERM) sh signal.sh

#nasilne ukonceni - nelze zachytit - signal SIGKILL
[]$ sh signal.sh &
[]$ ps a
PID  TTY      STAT   TIME COMMAND
9245 pts/0    S      0:00 sh signal.sh
[]$ kill -9 9245
[]$ ps a
...
[1]+  Zabit (SIGKILL) sh signal.sh
```

Výpis kódu 4.29: Rozdíl mezi signály pro nenásilné a násilné ukončení procesu.

Někdy je nepraktické posílat signály procesům pomocí jejich PID. Zde lze použít příkaz `pkill`, který zasílá signály procesům podle jejich názvů. Přepínač `-f` u příkazu `pkill` vyhledává zadaný text v příkazovém řádku[19], viz výpis 4.30.

```
[]$ sh signal.sh &
[]$ ps a
PID TTY      STAT   TIME COMMAND
9906 pts/0    S      0:00 sh signal.sh
[]$ pkill -9 -f signal.sh
[1]+  Zabit (SIGKILL) sh signal.sh
```

Výpis kódu 4.30: Zabití procesu dle jeho názvu.

[18]Rozdíl signálů SIGTERM a SIGINT (oba pro korektní ukončení procesu, mohou být zachyceny) je v tom, že SIGINT je zaslán pomocí znaků [Ctrl+c] v daném terminálu a SIGTERM pomocí příkazu `kill`. Rozlišuje se tedy zdroj signálu.

[19]Příkaz je zjednodušen, ať si nekomplikujeme život (není striktní při vyhledávání). Správná varianta je `pkill -9 -f sh[[:space:]]signal\.sh` pro přesnou shodu, protože příkaz očekává regulární výraz. Tečka v regulárním výrazu značí shodu s jakýmkoliv znakem, pro použití tečky v hledaném názvu je potřeba ji vyčlenit pomocí znaku `\`. Dále jsme vynechali část s `sh`, za kterou je mezera. Ta je v regulárním výrazu značena jako `[[:space:]]`.

4.9.4 Souběh procesů

Tento příklad demonstruje synchronizaci procesů při práci se sdílenou proměnnou. Situace, kdy dva nebo více procesů přistupují ke sdíleným prostředkům a výsledek závisí na pořadí a časovém okamžiku přístupu se nazývá souběh. Souběhu lze zabránit pomocí vyloučení možnosti čtení a zápisu sdílených dat ve stejném okamžiku. Do zdrojového kódu programu se přidá část kódu, která ovládá přístup ke sdíleným prostředkům, tato část kódu se nazývá kritická sekce. Pro vyloučení souběhu je zabráněno dvěma a více procesům vstoupit ve stejném čase do svých kritických sekcí.

Výpis 4.31 zobrazuje program se dvěma procesy, které pracují se sdíleným prostředkem – proměnnou typu interger. Hodnota této proměnné je nepředvídatelná, protože je nastavena podle toho, kdy dojde k přepnutí vykonávání procesů. Program vytvoří dva procesy A a B. Oba procesy pracují se stejnou proměnnou *sprom*. Vytvoření nových procesů je demonstrováno vytištěním jejich PID. Proces A hodnotu proměnné snižuje, proces B hodnotu proměnné zvyšuje. Proces zvyšování/snižování hodnoty je záměrně prodloužen smyčkou, aby mohlo dojít ke změně vykonávání procesu. Například když proces A snižuje proměnnou ve smyčce, dojde k přepnutí kontextu, a proces B začne ve smyčce proměnnou zvyšovat. Poté, co oba procesy dokončí svou činnost, tedy první proces sníží hodnotu o 1000 a druhý proces zvýší hodnotu o 1000, tak by výsledná hodnota proměnné měla být 0. To však není pravda, hodnota je náhodná podle stavu přepínání procesů – nastal stav souběhu.

```
1  import multiprocessing, os
2
3  def odeber(sprom):
4      print('Proces A ID:', os.getpid())
5      #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
6      for i in range(1000): sprom.value -= 1
7
8  def pridej(sprom):
9      print('Proces B ID:', os.getpid())
10     #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
11     for i in range(1000): sprom.value += 1
12
13 #sdilena promenna; typ promenne i -- signed int; init na 0
14 sprom = multiprocessing.Value('i', 0)
15 print('Pocatecni hodnota sdilene promenne '+str(sprom.value))
16
17 #vytvorim procesy
18 A = multiprocessing.Process(target=odeber, args=(sprom,))
19 B = multiprocessing.Process(target=pridej, args=(sprom,))
20
21 #spustim procesy a cekam nez dokonci praci
22 A.start(); B.start(); A.join(); B.join()
23
24 print('Konecna hodnota sdilene promenne '+str(sprom.value))
```

Výpis kódu 4.31: Práce se sdílenou proměnnou dvěma procesy

Souběh lze řešit několika způsoby. Jedním z nich je použití semaforů, které jsou nezávislé na programovacím jazyce. Semafor jako službu poskytuje operační systém a je

založen na předávání zpráv přes jádro systému. Výpis 4.32 zobrazuje použití semaforu pro opravu předchozího programu. V kódu je vytvořen semafor nazvaný *zamek*. Tento semafor je předaný jako parametr při vytváření procesů A a B. Před kritickou sekcí procesů A a B je aplikován zámek (semafor) nad sdílenou proměnnou. Zámek je po ukončení kritické sekce následně uvolněn. Po spuštění programu je výsledek sdílené proměnné vždy 0, což je správná hodnota.

```
import multiprocessing, os

def odeber(sprom,zamek):
    print('Proces A ID:', os.getpid())
    #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
    #zamknu a odemknu zamek nad sdilenou promennou
    zamek.acquire();
    for i in range(1000): sprom.value -= 1
    zamek.release()

def pridej(sprom,zamek):
    print('Proces B ID:', os.getpid())
    #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
    #zamknu a odemknu zamek nad sdilenou promennou
    zamek.acquire()
    for i in range(1000): sprom.value += 1
    zamek.release()

#sdilena promenna; typ promenne i -- signed int; init na 0
sprom = multiprocessing.Value('i', 0)
print('Pocatecni hodnota sdilene promenne '+str(sprom.value))

#vytvorim zamek
zamek = multiprocessing.Lock()

#vytvorim procesy
A = multiprocessing.Process(target=odeber, args=(sprom,zamek))
B = multiprocessing.Process(target=pridej, args=(sprom,zamek))

#spustim procesy a cekam nez dokonci praci
A.start(); B.start(); A.join(); B.join()

print('Konecna hodnota sdilene promenne '+str(sprom.value))
```

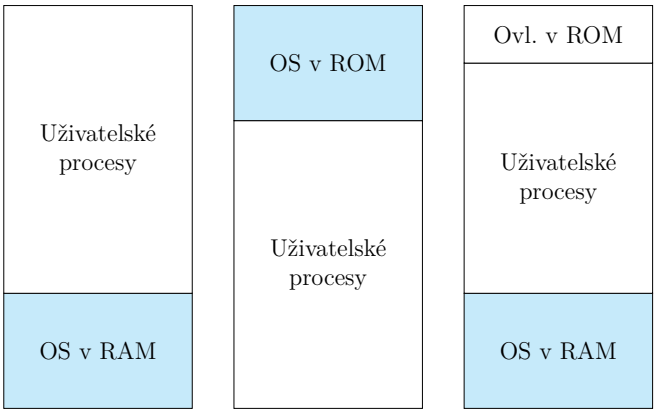
Výpis kódu 4.32: Použití semaforu při práci se sdílenou proměnnou dvěma procesy

5 PAMĚŤ

Operační systém přiděluje paměťový prostor procesům. Paměť dělíme na dvě základní oblasti:

- paměť pro operační systém,
- paměť pro uživatelské procesy.

Základní způsoby obsazení paměti jsou zobrazeny na obrázku 5.1. Operační systém může být umístěn v paměti RAM, ROM, nebo může být umístěn kombinovaně v RAM/ROM. V posledním případě jsou v ROM typicky ovladače. V paměti ROM se také nachází program BIOS (Basic Input Output System). Umístění celého systému do paměti ROM se používá u jednoúčelových (embedded) systémů [11].



Obrázek 5.1: Základní způsoby organizace paměti.

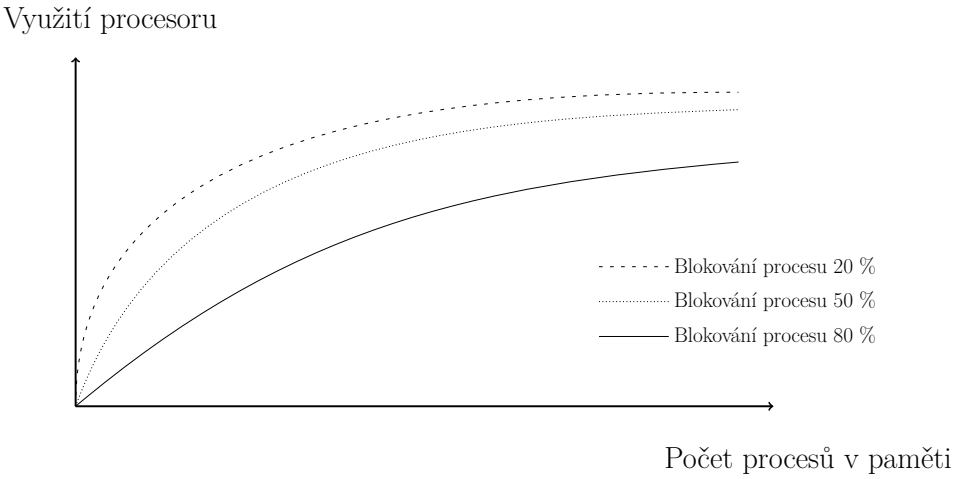
5.1 Dělení paměti pro procesy

V paměti se může nacházet jeden nebo více uživatelských procesů. Pokud je v paměti jeden uživatelský proces, tak je styl práce OS následující: Uživatel zadá příkaz, operační systém přesune žádaný program do paměti a spustí jej. Jak program ukončí svou činnost, tak operační systém vyzve uživatele k zadání dalšího příkazu. Do paměti je přesunut další program, který je spuštěn (tento program přepíše původní program). Pokud je v paměti pouze jeden proces a tento je blokován například I/O operací, tak procesor je v této době nevyužitý.

Při blokování jednoho procesu lze přidělit procesor jinému procesu. Je tak zvyšováno využití procesoru, který není necháván ležet „ladem“. Uvažme, že proces je blokován poměrem t z celkového času spuštění procesu. Máme n procesů v paměti. Pravděpodobnost n blokovaných procesů je t^n (procesor není využíván). Využití procesoru C můžeme vyjádřit pomocí vztahu $C = 1 - t^n$ [11].

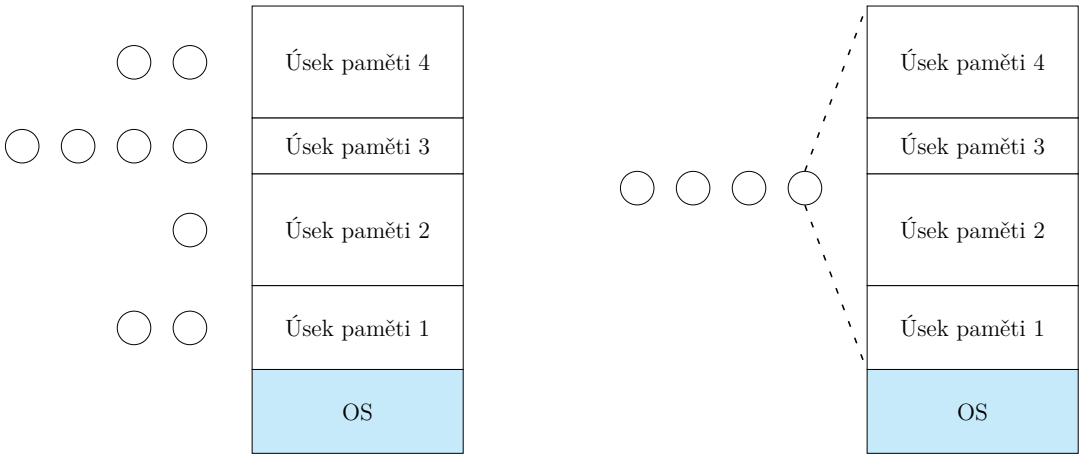
Využití procesoru pro n procesů je zobrazeno na obrázku 5.2. Pokud jsou procesy po 80 % svého času blokovány, tak by v systému muselo být přinejmenším 10 procesů, aby vytížení procesoru bylo cca 90 %. Uvedený výpočet je pouze aproximací, jelikož uvažuje,

že procesy jsou na sebe nezávislé. V reálném systému jsou však na sebe procesy závislé. Pro přesnější výpočty se proto používá teorie front.



Obrázek 5.2: Vytížení procesoru v závislosti na počtu procesů v paměti.

Jednoduchým způsobem řešícím umístění více procesů v paměti současně je trvalé rozdělení paměti na úseky při startu operačního systému. Tyto úseky paměti mohou mít různou **statickou** velikost a jsou přidělovány procesům[1]. Spuštěný proces je umístěn do čekací fronty pro úsek paměti, do kterého se celý proces vejde. Tato situace je zobrazena na obrázku 5.3. Použit je úsek paměti, ve kterém bude nejmenší plýtvání paměti, jelikož neobsazené místo v tomto úseku nelze využít pro jiné procesy.



Obrázek 5.3: Systém s více procesy v paměti.

Nevýhodou tohoto řešení je neoptimální využití paměti. Může nastat stav, kdy fronta pro úsek paměti s velkou velikostí je prázdná (fronta pro úsek 2) a fronta pro úsek s malou velikostí je zaplněna (fronty pro úseky 3, 4). Procesy s menším paměťovým nárokem[2] musí

[1] Paměťovému prostoru procesu.
[2] Malým paměťovým prostorem.

čekat ve frontě na svůj úsek, i když jsou k dispozici úseky o větší velikosti. Alternativou je použití jedné fronty pro všechny procesy. Pokud je nějaký úsek paměti volný, tak je přidělen prvnímu procesu ve frontě, který se zde vejde. Toto řešení přináší značné plýtvání paměti, protože procesy s malým paměťovým nárokem mohou být umístěny do velkých úseků paměti. Možná je modifikace, kdy je prohledána fronta procesů a je vybrán proces s největším paměťovým nárokem, který se do volného úseku paměti vejde. Toto řešení preferuje větší procesy nad malými. Zde je řešením použít pravidlo, že proces ve frontě může být přeskočen maximálně k krát. Po vyčerpání limitu přeskočení je proces umístěn do bloku paměti bez ohledu na jeho velikost.

Operační systémy mohou také paměť dělit na fixní (statické) bloky, které jsou přidělovány dynamicky. Jedná se o operační systémy reálného času, jelikož doba alokace bloku paměti fixní velikosti je konstantní a tudíž předvídatelná (viz požadavky na systémy reálného času, kapitola 4.5.2). Příkladem může být operační systém µC/OS-III, který se používá v leteckých systémech.

Pokud je velikost alokované paměti dynamická – odpovídá velikosti paměťového prostoru procesu (uživatelský kontext) – dostáváme se k plnohodnotné **dynamické** alokaci. Úseky paměti o různé velikosti mohou být vytvářeny a uvolňovány tak, že mezi nimi budou části paměti nevyužité, jako je například zobrazeno na obrázku 5.4. Vzniká tak externí fragmentace volné paměti. Nevyužitý prostor je možno spojit do jednoho většího úseku, který pak je k dispozici pro alokaci procesům s větším paměťovým nárokem. Tento proces se nazývá **slučování volné paměti** (Memory Compaction). Pro tento úkon je typicky potřeba přesunout všechny procesy v paměti na nové umístění. V praxi se však toto setřepání volné paměti nepoužívá z důvodu náročnosti provedení. Uvažme stroj s pamětí 256 MiB, který kopíruje 4 B za 40 ns. Přesun celé paměti zabere okolo 2,7 sekundy. Tuto akci by bylo nutno provádět při každé nové alokaci nebo ve zvolených intervalech. Operační systém by tedy byl prioritně zaneprázdněn pouze seskupováním volné paměti.

Obrázek 5.4: Volné úseky paměti při dynamické alokaci.

Při alokaci paměti je potřeba pracovat a tím, že procesy mění svou velikost – žádají o paměť a tu pak uvolňují (halda, zásobník). Mohou nastat tyto dvě situace:

- V sousední části paměti je volný prostor; lze přidělit procesu při jeho růstu.
- Proces sousedí v paměti s jiným procesem. Celý proces je potřeba přesunout na nové místo v paměti, které jeho rozšíření umožní. Toto je časově náročná operace.

Řešením je při alokaci paměti pro proces zabrat rezervu, jak je zobrazeno na obrázku 5.5.

Obrázek 5.5: Dynamická alokace s rezervou.

Při přidělování paměti procesům může nastat situace, kdy se všechny procesy nevejdou do hlavní paměti. V tomto případě je využit koncept **virtuální paměti** (VM – virtual memory), kdy procesy jsou vyměňovány mezi **hlavní** a **odkládací** (vedlejší) pamětí.

Při popisu virtuální paměti se používají tyto pojmy:

- **Fyzický adresový prostor (FAP)** (memory space, hlavní paměť, operační paměť, paměť RAM) – skutečná fyzická paměť. Jeho rozsah je dán kapacitou instalované paměti. Aby proces mohl být vykonáván, musí být celý nebo jeho část umístěna ve fyzické paměti. Důvodem je vyšší rychlost přístupu k této paměti.
- **Logický adresový prostor (LAP)** (address space) – paměť z pohledu procesů. Je určen adresami, které je proces (operační systém) schopen generovat. LAP zahrnuje fyzický adresový prostor a **odkládací prostor** (vedlejší paměť), viz obrázek 5.7.

Například počítač s 32bitovým CPU, nabízí LAP o velikosti 4 GiB (2^{32}), ačkoliv fyzicky disponuje jedním 256 MiB paměťovým modulem (FAP).

Pokud by byl dostatek hlavní paměti, tak by použití virtuální paměti nebylo potřeba. Je tedy otázka budoucnosti, zda budou k dispozici tak velké fyzické operační paměti. Na druhou stranu nároky software na paměť rostou rychleji než technologický vývoj operačních pamětí. Proto systém virtuální paměti bude pravděpodobně stále potřeba.

Virtuální paměť lze realizovat několika způsoby:

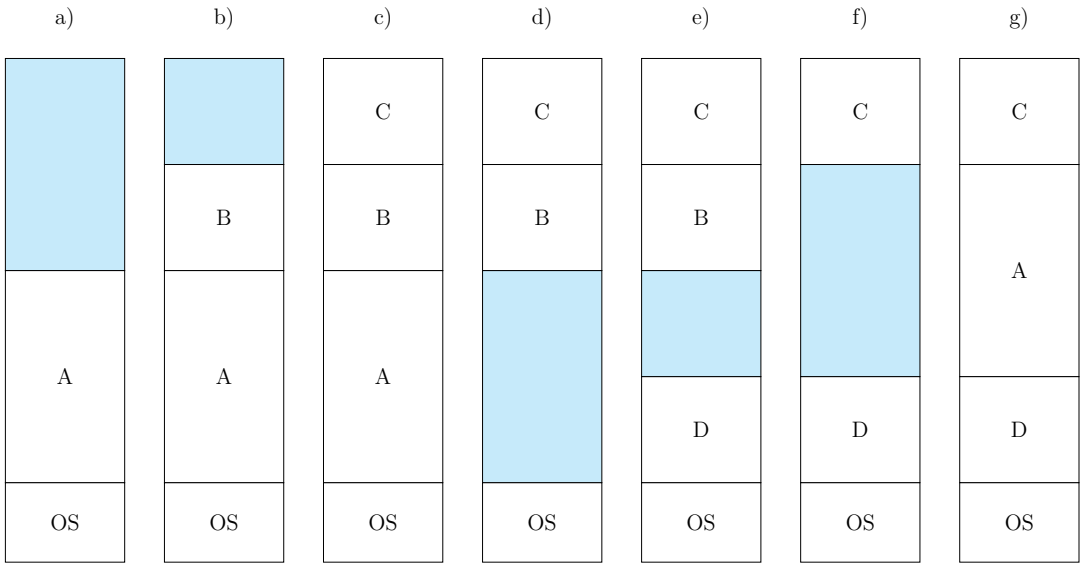
- vyměňováním celých procesů,
- stránkováním,
- segmentací.

5.1.1 Vyměňování procesů

Hlavní paměť lze přidělovat celým procesům nebo jen jejich částem. Celé procesy nebo jejich části jsou pak vyměňovány mezi hlavní a odkládací pamětí.

Pokud je nutno proces vykonávat a není v hlavní paměti, je celý proces přenesen z odkládací paměti. Po určité době je přesunut do odkládací paměti. Procesy jsou vyměňovány a každý proces stráví určitou dobu v hlavní paměti, aby bylo umožněno jeho vykonávání.

Příklad vyměňování celých procesů je uveden na obrázku 5.6 [11]. Nejprve je v hlavní paměti pouze proces A. Následně jsou do hlavní paměti přeneseny procesy B a C. Po nějaké době je proces A odložen a vznikne místo pro umístění procesu D (proces D má menší velikost než proces A). Dále je do odkládací paměti přesunut proces B. Následně se zpátky do hlavní paměti vrací proces A, ovšem na jiné adrese.



Obrázek 5.6: Příklad změn v hlavní paměti při vyměňování celých procesů.

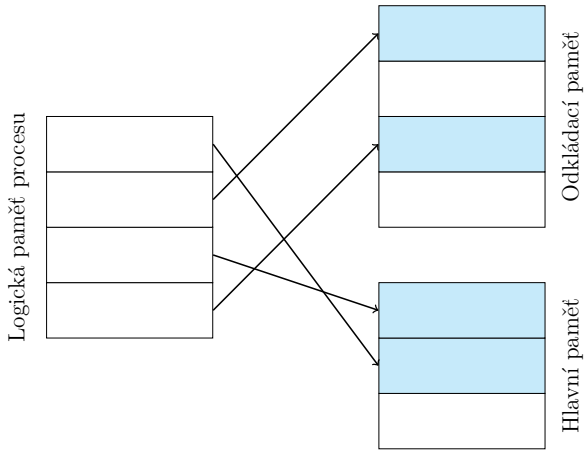
Umístění procesů v hlavní paměti se mění podle toho, jak jsou odkládány. Tím je způsobena **externí fragmentace** volné paměti. Vznikají volná místa v hlavní paměti, která nelze použít pro procesy s větší velikostí. Oblasti volné paměti je možné sloučit a vytvořit tak větší prostor. Prakticky se však neprovádí pro časovou náročnost, viz předchozí popis.

Koncept virtuální paměti umožňuje běh procesů, i když nejsou celé v hlavní paměti. Paměťový obraz procesu je rozdělen na části (overlays). Proces lze začít vykonávat, pokud má svou první část v hlavní paměti. Pro další činnost je přenesena do hlavní paměti druhá část atd. V paměti lze umístit i více částí procesu najednou. Nejdříve to byl programátor, kdo dělil program na jednotlivé části pro jejich umístění v paměti. Následně začal dělení procesu na části provádět operační systém.

Například jeden proces o velikosti 16 MiB může být vykonáván na stroji, který má k dispozici pouze 4 MiB hlavní paměti[3]. Systém uloží do hlavní paměti pouze ty části procesu, které jsou v daný okamžik potřeba pro jeho vykonávání. Druhým příkladem je počítač, který pracuje se 16bitovou adresací, LAP má tedy velikost 64 KiB. Ovšem hlavní paměť má velikost pouze 32 KiB. Zde může běžet proces s velikostí až 64 KiB (neuvažujeme paměť pro OS).

Příklad rozložení paměťového prostoru jednoho procesu mezi hlavní a odkládací pamětí je zobrazen na obrázku 5.7.

[3]Uvedené hodnoty jsou ilustrativní.



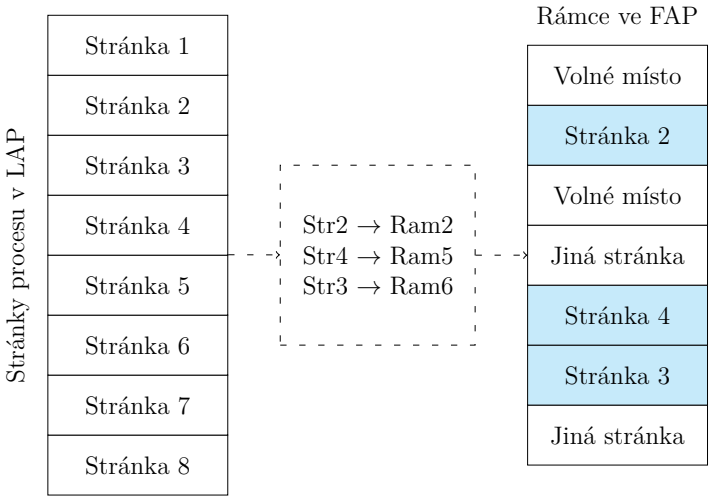
Obrázek 5.7: Koncept virtuální paměti.

V případě více procesů operační systém udržuje v hlavní paměti jen určité části. Zbylé části jsou uloženy v odkládací paměti. Operační systém části procesů přesunuje (kopíruje) do/z hlavní paměti podle potřeby. Paměťový obraz procesů se dělí na **stránky** a **segmenty**.

5.1.2 Stránkování

U stránkování je paměťový prostor procesu rozdělen na stejně velké úseky – **stránky** (pages). Stránky jsou přesunuty[4] z odkládací paměti do **rámců** (frames) v hlavní paměti, které mají stejnou velikost. Stránka je přesunuta do hlavní paměti, když vykonávaný proces potřebuje pracovat s touto stránkou – nastává **výpadek stránky** (page fault). Tuto skutečnost rozpozná procesor (nikoli proces). Rámec může být prázdný, nebo je obsah zapsán do odkládací paměti. Výpadek stránky a jeho ošetření je pro proces neviditelný. Viditelný je ovšem pro operační systém, který vede evidenci umístění stránek v **tabulce stránek** (page table), která je zobrazena na obrázku 5.8 [16].

[4]Slovo „přesunuty“ zatím používáme pro zjednodušený výklad. Ve skutečnosti je to „kopie s možnou aktualizací v odkládací paměti“. Později bude vysvětleno blíže.



Obrázek 5.8: Tabulka stránek.

Výhoda stránkování je, že díky jejich pevné velikosti (např. 4 KiB a 64 KiB) je lze efektivně umístit do stejně velkých rámců. Při velikostech LAP 64 KiB, FAP 32 KiB a stránky 4 KiB je k dispozici 16 stránek a 8 rámců. Nedochází tak k fragmentaci volného prostoru v hlavní paměti. Nevýhodou je, že proces nemá možnost určit, které stránky logicky patří k sobě, a tak může docházet k častějším výpadkům stránek. Při přesunu stránky by mohl proces určit, které další stránky bude vzápětí potřebovat a přesunout tyto stránky do hlavní paměti společně.

Důležitým parametrem stránkování je volba velikosti stránky. Velikost stránky je určena na základě několika faktorů. Začněme nejprve s důvodu pro malou velikost stránek. Při použití větších stránek bude docházet k situacím, kdy poslední stránka nebude zcela zaplněna. Volný prostor v rámci poslední stránky pak nelze využít pro jiná data. Tento stav se nazývá **interní fragmentace** [11].

Další faktor mluvící pro malou stránku si ukážeme na příkladě. Uvažme program zahrnující několik fází, které jdou sekvenčně po sobě. Každá fáze zabírá 4 KiB paměti. Při velikosti stránek 32 KiB má program přiděleno 32 KiB hlavní paměti po celou dobu běhu. Pokud by velikost stránek byla 16 KiB, program by měl alokováno 16 KiB paměti. Při velikosti stránky 4 KiB by program potřeboval pouze 4 KiB hlavní paměti.

Na druhou stranu, malá velikost stránky znamená jejich větší počet, a tudíž větší tabulku stránek. Přitom čas přenesení stránky z a do odkládací paměti je tvořen zejména časem pro vyhledání stránky v paměti. Vlastní doba přenosu malých stránek je přibližně stejná jako pro velké stránky. Příkladem může být čas pro přenos 64 stránek o velikosti 512 B daný výpočtem $64 \times 10 = 640$ ms, zatímco přenos 4 stránek o velikosti 8 KiB může trvat $4 \times 12 = 48$ ms [11].

5.1.3 Segmentace

Segmentace dělí paměťový prostor procesu na logické části – **segmenty**, které mají různou velikost. Segmenty vytváří kompilátor podle zdrojového kódu programu. Pojetí segmentů při správě paměti se liší – obecné segmenty mohou například být:

- Textový – instrukce programu.
- Datový – inicializované globální a statické proměnné.
- BSS (Block Started by Symbol) – neinicializovaná data; mohou mít náhodnou hodnotu nebo hodnotu „0“ (v případě ukazatele „null“).
- Halda – dynamická alokace.
- Zásobník.

Příklad velikostí obecných statických segmentů programů `ping` a `python` je uveden ve výpisu 5.1.

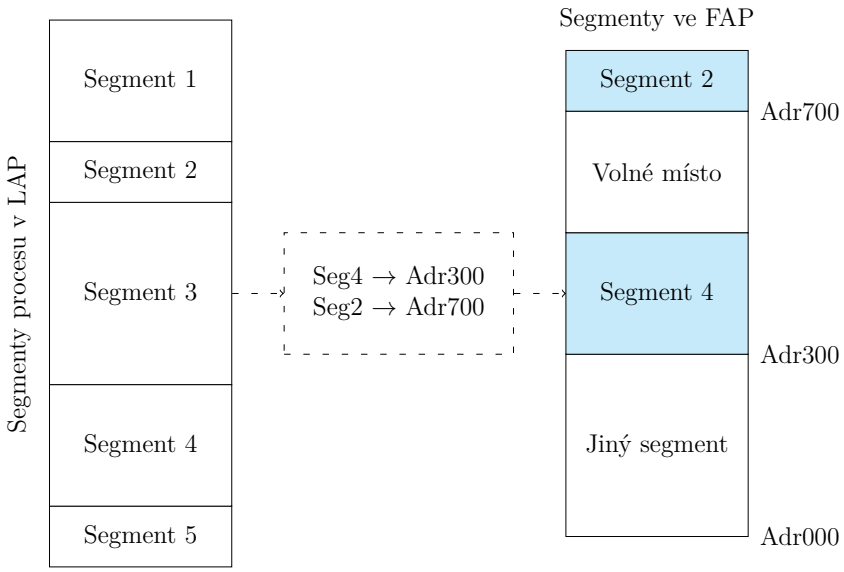
```
[]$ size /usr/bin/ping /usr/bin/python
text    data   bss      filename
55324   2440   145536   /usr/bin/ping
1859    628    4        /usr/bin/python
```

Výpis kódu 5.1: Příklad statických segmentů pro dva programy.

Bližší dělení segmentů pak například může být na:

- kódy funkcí,
- velké datové struktury,
- knihovny linkované s programem,
- objekty.

Na rozdíl od stránkování je mechanismus segmentace pro proces viditelný. Pokud je nějaká část segmentu potřebná v hlavní paměti, tak je přenesen celý segment. Podobně jako u stránkování, operační systém vede evidenci umístění segmentů v hlavní a odkládací paměti v **tabulce segmentů** (segment table), která je zobrazena na obrázku 5.9 [16].



Obrázek 5.9: Tabulka segmentů.

Výhodou segmentace oproti stránkování je minimalizace počtu výpadků segmentů. Nevýhodou je ztráta efektivnosti při umisťování segmentů do hlavní paměti – segmenty nemají pevnou velikost a způsobují fragmentaci hlavní paměti.

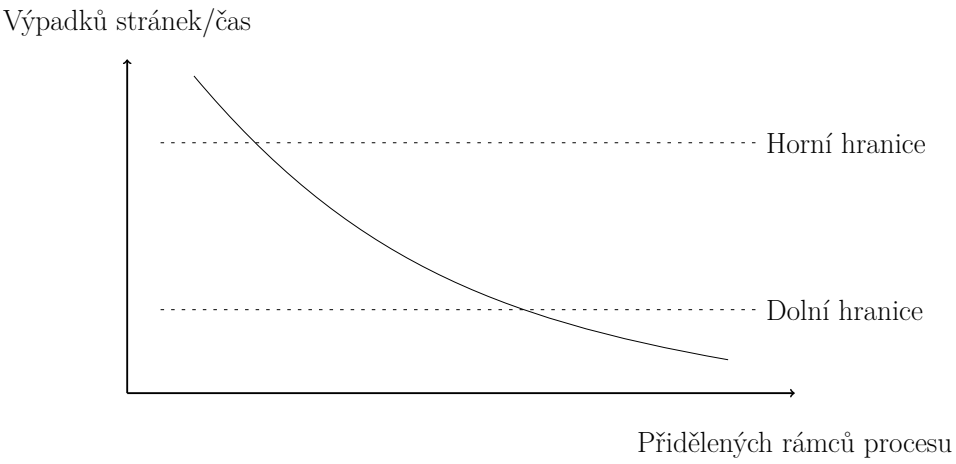
Stránkování i segmentace mají své výhody i nevýhody. Některé architektury kombinují oba způsoby.

5.2 Alokace paměti

Přidělovat hlavní paměť procesům lze několika způsoby. Jednoduchým způsobem je přidělovat hlavní paměť rovnoměrně. Ovšem toto řešení není dobré, protože nerespektuje vlastnosti procesů, například velikost jejich paměťového prostoru. Dalším možným přístupem je uvážit celkovou velikost procesu a každému procesu přidělit počet rámců proporcionálně k počtu jeho stránek. Obě tyto metody nerespektují prioritu procesů. Přitom procesům s vyšší prioritou by mělo být přiděleno více rámců v hlavní paměti pro urychlení jejich práce – proces nebude čekat na provedení častých výpadků stránek. Další způsob tedy je, že při výpočtu rámců pro přidělení jednotlivým procesům je zvažována jejich priorita. Ovšem priorita procesů se může měnit.

Pro určení vhodného počtu rámců pro přidělení procesům lze také použít metodu založenou na frekvenci výpadků stránek. Smyslem je vyhnout se stavu, kdy je výpadků stránek mnoho a proces je zdržován přesunem stránek z odkládací do hlavní paměti (provádí OS). Tento stav se nazývá **trashing** [16]. Z tohoto důvodu je potřeba „držet“ počet (frekvenci) výpadků stránek pro všechny procesy na přibližně stejné úrovni. Jestliže je frekvence výpadků pro proces velká, je to signál, že proces potřebuje více rámců v hlavní paměti – s větším počtem rámců přidělených procesu frekvence výpadků klesne. Na druhou stranu, pokud je frekvence výpadků stránek nízká, je to signál, že proces má příliš mnoho přidělených rámců v hlavní paměti. Tyto rámce mohou být přiděleny jiným procesům, u kterých by došlo ke snížení počtu výpadků stránek.

Tento princip byl zaveden algoritmem nazvaným **Page Fault Frequency Replacement** [29]. Příklad závislosti frekvence výpadků stránek procesu na počtu přidělených rámců je zobrazen na obrázku 5.10 [16]. Obrázek zobrazuje horní a spodní hranici, které jsou signály pro přidělení/odebrání hlavní paměti (rámců) danému procesu.



Obrázek 5.10: Princip algoritmu Page Fault Frequency Replacement.

5.3 Přesuny stránek

Virtuální paměť je založena na přesunu (kopírování) stránek mezi hlavní a odkládací pamětí. Když je proveden výpadek stránky (vykonávaný proces vyžadoval data ve stránce, ale stránka byla ve vedlejší paměti), tak je stránka kopírována do volného rámce v hlavní paměti. Pokud žádný rámec není volný, je zvolena stránka, která bude z rámce přesunuta do odkládací paměti. Tím se rámec uvolní pro novou stránku. Toto řešení ad-hoc (v případě potřeby) je pomalé. Proto operační systém periodicky prohledává hlavní paměť a přesunuje určitý počet stránek z rámců do odkládací paměti. Tímto je stále udržován určitý počet volných rámců, které jsou ihned připraveny pro zaplnění stránkami.

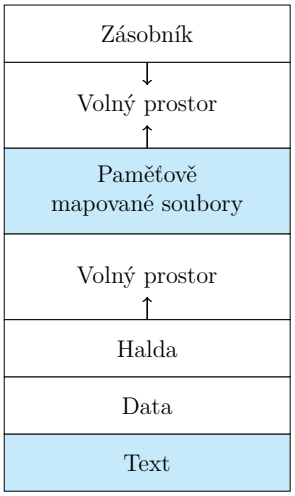
5.3.1 Odkládací paměť

Při odložení stránky z FAP mohou nastat dvě varianty podle stavu dat ve stránce – i) data jsou pouze pro čtení nebo nebyly provedeny změny – stránka se jen z rámce smaže, ii) data byla pozměněna – stránka je přesunuta do odkládací paměti. Odkládací pamětí může být:

- soubor stránky,
- odkládací prostor.

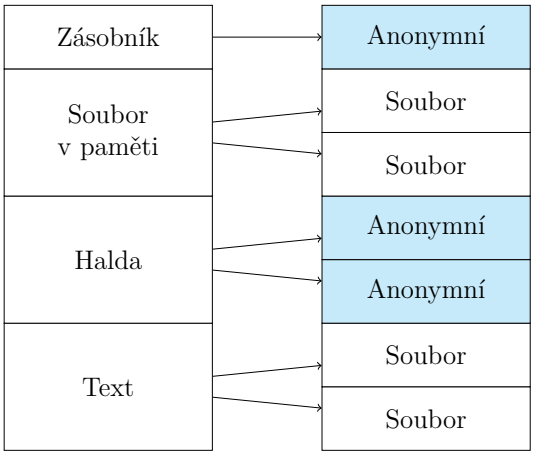
Stránky mohou být svázané s určitým souborem. Takový soubor může obsahovat programový kód procesu (data pouze pro čtení) nebo se může jednat o soubor připojený do uživatelského kontextu procesu, tzv. **paměťově mapovaný soubor**. Příklad rozložení uživatelského kontextu je zobrazen na obrázku 5.11. Soubor může být mapovaný v režimu pouze pro čtení nebo i pro zápis. K těmto souborům je přistupováno pomocí funkcí pro práci s pamětí, nepoužívají se tedy funkce pro práci se soubory v souborových systémech. Tento způsob se také používá pro sdílení jednoho souboru mezi více procesy. Výhodou je vyšší rychlost čtení a zápisu. Nevýhodou je pak plýtvání pamětí. Jednotkou pro práci s daty je jedna stránka. Pokud je velikost stránky 4 KiB, tak soubor o velikosti 6 KiB zabere stránky dvě a zbylé místo v druhé stránce je nevyužité. Používá se také pro sdílení

souboru mezi více procesy. V případě změn dat v těchto stránkách je potřeba tyto změny provést i ve svázaném souboru (při jejich odložení).



Obrázek 5.11: Uživatelský kontext s částmi, které jsou svázané se specifickými soubory.

Stránky, které nejsou svázané se souborem v souborovém systému, jsou nazývány **anonymní** a jsou přesunuty do **odkládacího souboru** (swap file) v souborovém systému nebo **odkládacího oddílu** (swap partition)[5] na úložném zařízení. Příklad anonymních stránek je zobrazen na obrázku 5.12. Jedná se o části uživatelského kontextu data, haldy a zásobníku.



Obrázek 5.12: Uživatelský kontext a anonymní stránky.

5.3.2 Algoritmy výběru stránek

Operační systém udržuje určitý počet rámců v hlavní paměti stále volný, aby se do nich při výpadku stránky mohly ihned umístit stránky z odkládací paměti. Pro tento účel

[5]Záleží na implementaci podle operačního systému.

je prováděna periodická kontrola počtu volných rámců. Pokud jejich počet klesne pod určitou hranici, jsou zvolené stránky přesunuty do odkládací paměti.

Které stránky se mají z rámců přesunout? Možné řešení je stránky k přesunu vybírat náhodně. Ovšem náhodně vybraná stránka může být vzápětí potřebná procesem a bude zpět přesunuta do hlavní paměti. Tyto akce jsou zbytečné a zpomalují běh operačního systému. Ideální je vybrat ty stránky k přesunu, které budou procesem použity až za delší dobu.

Tento problém se netýká pouze přesunu stránek, ale zasahuje do mnoha dalších oblastí. Uveďme si příklad webového serveru (příklad je podán zjednodušeně bez uvažování práce OS). Tento server může ve své vyrovnávací paměti udržovat určitý počet často používaných webových stránek. Když se tato paměť zaplní a je potřeba zobrazit novou stránku (přesunout ji do vyrovnávací paměti), tak se musí nějaká stránka z vyrovnávací paměti odstranit. Náhodný výběr by mohl zvolit stránku, která bude vzápětí znovu potřebná ve vyrovnávací paměti. Pokud by byla k přesunu vybrána webová stránka, která bude použita až za nějakou dobu, tak server bude reagovat rychleji.

Jakou stránku tedy vybrat k přesunu z rámce v hlavní paměti, aby počet výpadků stránek byl minimální? Uvažme tuto situaci. Je potřeba přesunout některé stránky, aby se uvolnily rámce. Jedna z těchto stránek může být použita při vykonávání další instrukce aktuálního procesu, další nemusí být použity delší dobu (za určitý počet vykonaných instrukcí). Ideální je vybrat stránku (nebo stránky), které budou použity za největší počet instrukcí (tj. nejdále v budoucnu). Pokud najdeme takovéto stránky, tak máme vyhráno, protože počet výpadků stránek bude minimální.

Řešením je ke každé stránce přidružit počet instrukcí, které budou vykonány před jejím použitím. Zvolena je pak stránka s nejvyšším počtem instrukcí. Problém však je v tom, že operační systém nemůže určit tyto počty instrukcí z důvodu, že neumí předpovídat budoucnost, tj. kdy a která událost nastane[6]. Zde by mohla pomoci dobrá věštecká skleněná koule.

Této oblasti bylo věnováno hodně výzkumu a existují různá řešení. Podívejme se na některá základní, které se tomu ideálnímu více či méně blíží [11].

Algoritmus LRU (Least Recently Used)

Aproximaci ideálního způsobu získáme úvahou, že stránky, které byly často používány během předchozích instrukcí, budou patrně často používány i během instrukcí následujících. A naopak, stránky dlouho nepoužívané se pravděpodobně nebudou používat i nadále. Základem algoritmu LRU je tedy myšlenka „odlož stránku, která nejdéle leží ladem“. Pro jeho realizaci je nutno udržovat pořadový seznam všech stránek v hlavní paměti podle jejich doby použití. První v seznamu je posledně použitá stránka a poslední je nejdéle nepoužitá. Seznam je aktualizován při použití každé stránky. Nalezení stránky v seznamu, její smazání na současném místě a přesun na první místo je časově náročná operace.

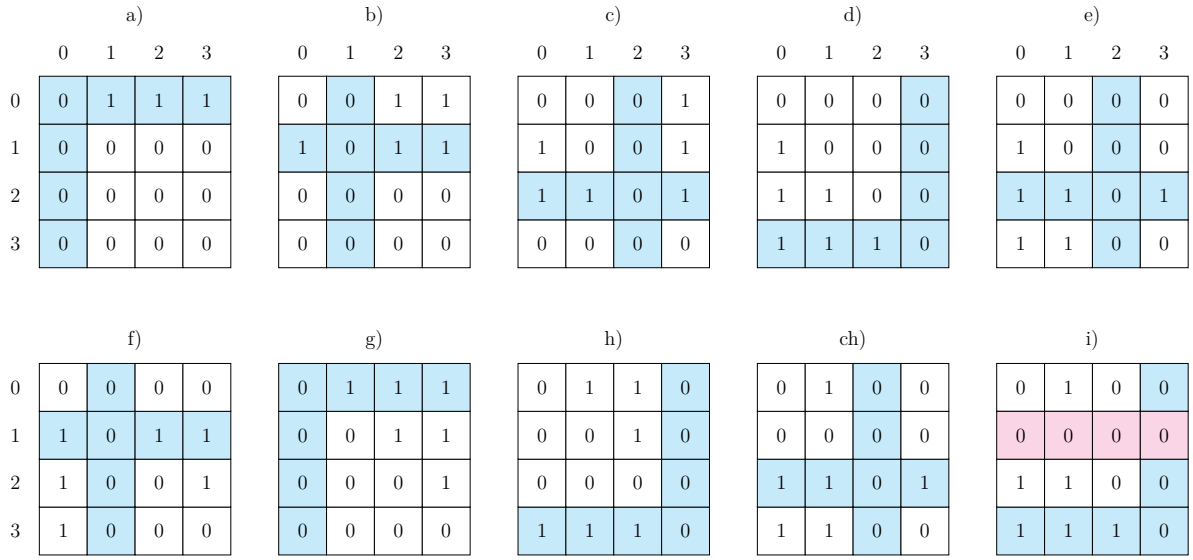
[6] Výjimkou mohou být programy, které běží pouze samy a nejsou nikdy přerušeny nějakou externí událostí. V tomto případě lze program spustit poprvé a pro každou stránku v paměti monitorovat, kdy byla použita. Tak lze určit instrukce, které stránku vyžadují a pořadí těchto instrukcí. Tyto počty instrukcí pak přiřadit stránkám. Při druhém spuštění programu lze pak určit, která stránka bude použita až za největší počet vykonaných instrukcí.

Pro rychlé provedení tohoto algoritmu lze použít hardware. Představme si dva základní způsoby.

První spočívá v použití hardware se 64bitovým čítačem. Tento čítač je inkrementován po vykonání každé instrukce. Každá stránka má přidruženo návěští, které obsahuje hodnotu tohoto čítače. Po „použití“ stránky se do návěští uloží aktuální hodnota čítače. Když je potřeba odstranit nějakou stránku, je vybrána ta s nejnižší hodnotou čítače (jedná se o nejdéle nepoužitou stránku).

Další způsob spočívá ve využití hardware, který pracuje s maticemi. Pro n stránek v hlavní paměti je použita matice $n \times n$ bitů. Na počátku jsou všechny bity nulové. Kdykoliv je použita stránka k, jsou nejdříve nastaveny bity řádku k na 1 a poté bity sloupce k na 0. Platí, že řádek s nejnižší binární hodnotou patří stránce nejdéle nepoužité. Vybrána bude tedy tato stránka.

Na obrázku 5.13 [11] je znázorněn příklad pro 4 stránky v hlavní paměti. K stránkám bylo přistoupeno v pořadí 0, 1, 2, 3, 2, 1, 0, 3, 2, 3. Nejdéle nepoužitou je stránka číslo 1, čemuž odpovídá na obrázku stav i). Naopak poslední použitá stránka je č. 3, což opět koresponduje se stavem i).



Obrázek 5.13: LRU pro stránky v pořadí 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Algoritmus NFU (Not Frequently Used) a „Aging“

Softwarový algoritmus NFU staví na použití čítače, podobně jako první hardwarová implementace LRU [11]. Tento čítač je přiřazen každé stránce a má počáteční hodnotu nula. Pro každou stránku je udržován bit A (accessed)[7], který je nastaven když je daná stránka použita. Použití stránky je monitorováno v intervalech daných přerušením od hodin. Při každém intervalu jsou všechny stránky prohledány a hodnota bitu A je přidána k hodnotě programového čítače. Přidána je hodnota 0, pokud ke stránce nebylo přistoupeno.

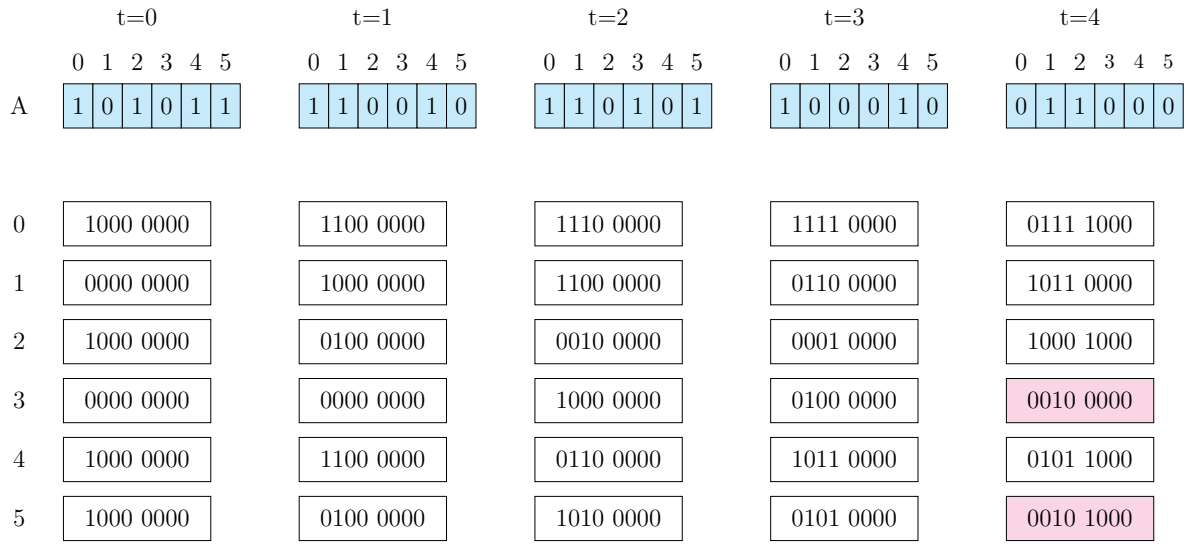
[7] Označováno také jako R (referenced).

Hodnota 1 je přidána, pokud ke stránce přistoupeno bylo. Při potřebě přesunu stránky je vybrána stránka s nejnižší hodnotou čítače.

Nevýhodou algoritmu NFU je, že „nezapomíná“. To znamená, že v minulosti často používané stránky mají velkou hodnotu čítače. Ačkoliv se tyto stránky již dlouho nepoužívají, tak je algoritmus k odstranění nevybere. Modifikovaný algoritmus má příhodný název stárnutí stránky (page aging).

Změny oproti algoritmu NFU jsou dvě. Čítače stránek jsou před přičtením hodnoty bitu A nejdříve bitově posunuty o 1 místo vpravo (dělení dvěma). Následně je bit A přičten na místo s nejvyšší binární váhou, MSB (Most Significant Bit)[8]. U NFU to byl bit LSB (Least Significant Bit).

Práci algoritmu lze pochopit z obrázku 5.14 [11]. Dejme tomu, že po prvním intervalu hodin $t = 0$ mají bity A u stránek 0 až 5 hodnoty 1, 0, 1, 0, 1, 1, tj. stránka 0 má $A = 1$, stránka 1 má $A = 0$ atd. Jinými slovy, během časového intervalu $t = 0$ byly použity stránky 0, 2, 4 a 5, zatímco ostatní stránky byly nepoužity. Tato situace je zobrazena v prvním sloupci. Na obrázku je ve sloupcích uveden stav poté, co byly čítače bitově posunuty a bit A přičten. Další sloupce zobrazují stav při následujících časových intervalech.



Obrázek 5.14: Algoritmus stárnutí. Pět hodinových cyklů, šest stránek paměti.

Uvažme stav v posledním sloupci po intervalu $t = 4$. Z obrázku je vidět, že stránky 3 a 5 nebyly použity v intervalech $t = 4$ a $t = 3$. Zároveň bylo k oběma přistoupeno v intervalu $t = 2$. Má-li být odstraněna nějaká stránka, bude se tedy rozhodovat mezi stránkou 3 a 5. Nelze ovšem určit, která z těchto dvou stránek byla použita později v rámci intervalu $t = 2$. Odstraněna bude stránka č. 3, protože má nižší hodnotu čítače (ke stránce č. 5 bylo přistoupeno v intervalu $t = 0$). Tato stránka nemusí být ovšem tou nejdéle nepoužitou.

[8]Nejvýznamnější bit, bit s největší váhou; tedy ten nejvíce vlevo.

Dalším nedostatkem algoritmu stárnutí je, že čítače mají omezený počet bitů, v našem ukázkovém příkladě 8. Pokud mají dvě a více stránek hodnotu čítače 0, tak nelze říci, kdy byla která stránka použita. V tomto případě je proveden výběr stránky náhodně. Jedná se o obecný problém přetečení čítačů.

Aplikace algoritmů

Uvedené algoritmy obecně řeší použití omezeného rychlého zdroje (RAM) a „neomezeného“ pomalého zdroje (odkládací prostor). U omezeného zdroje je potřeba optimálně vybírat data pro přesun, aby byl uvolněn prostor. Náhodně vybraná data mohou být vzápětí potřebná a zpět přesunuta, což nechceme. Tento problém se prolíná do různých oblastí, kde se tyto typy zdrojů vyskytují. Již byl uveden příklad webového serveru, kde se ve vyrovnávací paměti udržuje určitý počet často používaných stránek tak, aby se tyto stránky nemusely stále načítat z disku.

Další příklad je založen na finančním dopadu. IP geolokace je název procesu pro získání geografické polohy IP adresy. Je zaslán dotaz na vzdálenou geolokační databázi, která následně vrátí pozici pro zadanou adresu[9] ve formě země, města a souřadnic. Každý dotaz na polohu je za poplatek[10]. Při velkém počtu stálých dotazů, například na polohu návštěvníků webového portálu, se ročně může jednat o významnou položku. Je vhodné předešle provedené dotazy ukládat do vyrovnávací paměti a v případě dotazu na polohu stejné adresy – vracející se návštěvník – tuto polohu z vyrovnávací paměti použít. Při zaplnění paměti je nutné vybrat položku, která bude odstraněna, a tak uvolněno místo pro nový záznam.

Pro programovací jazyky jsou dostupné nástroje, které pracují s vyrovnávací pamětí. LRU algoritmus je implementován funkcí `lru_cache()` v jazyce Python. Účelem této funkce je zrychlit výpočetní operace pomocí ukládání výsledků předešlých výpočtů. Aplikace je zobrazena ve výpisu 5.2 [36]. Uvedený kód provádí rekurzivní výpočet Fibonačiho posloupnosti, kde každé číslo této posloupnosti je součtem dvou čísel předcházejících tomuto číslu $F(n) = F(n-1) + F(n-2)$, definičním oborem funkce jsou přirozená čísla; pro $n = 0$ a $n = 1$ je výsledek definován jako n. Původně byla tato posloupnost použita pro popis (idealizovaného) rozmnožování králíků, kde vstupním parametrem n byl počet měsíců. Ve výpisu je klíčový je řádek `@lru_cache(maxsize=100)`, pomocí které funkce pracuje s vyrovnávací pamětí LRU; funkce `fib_with_cache` je předána jako vstup do funkce `lru_cache`. Parametr `maxsize` určuje velikost této paměti (bez parametru by vyrovnávací paměť rostla donekonečna). Výstupem je porovnání času provedení funkce pro výpočet posloupnosti s použitím vyrovnávací paměti a bez ní. Se zvyšujícím počtem rekurzí rozdíl časů roste. Algoritmus LRU řeší, který výsledek výpočtu vyřadit z vyrovnávací paměti, pokud dojde k jejímu zaplnění.

```
1 from functools import lru_cache; import time
```

[9] Geolokační databáze může být nainstalována i lokálně. Databáze poskytované pro lokální instalaci mají nižší přesnost odhadu polohy. Pro získání přesnějších poloh se používá placený dotaz na vzdálenou databázi, která je provozována geolokační službou. Lokálně instalovanou databázi je potřeba aktualizovat, protože adresní prostor Internetu se neustále mění.

[10] Přibližně tisícinu dolaru a méně za dotaz, dle počtu požadovaných informací – poloha, ISP, AS, typ připojení atd.

```
2
3 #vypocet Fibonaciho cisla
4 def fib_without_cache(n):  #bez LRU
5  if n < 2:
6   return n
7  return fib_without_cache(n-1)+fib_without_cache(n-2)
8
9 begin = time.time()
10 fib_without_cache(45) #pocet iteraci 35(3s) 40(32s) 45(213s)
11 end = time.time()
12 print(end-begin) #cas vypoctu bez LRU
13 #213 s
14
15 @lru_cache(maxsize=128)
16 def fib_with_cache(n): #s LRU
17  if n < 2:
18   return n
19  return fib_with_cache(n-1) + fib_with_cache(n-2)
20
21 begin = time.time()
22 fib_with_cache(45) #pocet iteraci 35(0s) 40(0s) 45(0s)
23 end = time.time()
24 print(end-begin) #cas vypoctu s LRU
25 #0.06 ms
```

Výpis kódu 5.2: Aplikace vyrovnávací paměti s algoritmem LRU pro zrychlení provedení rekurzivní funkce.

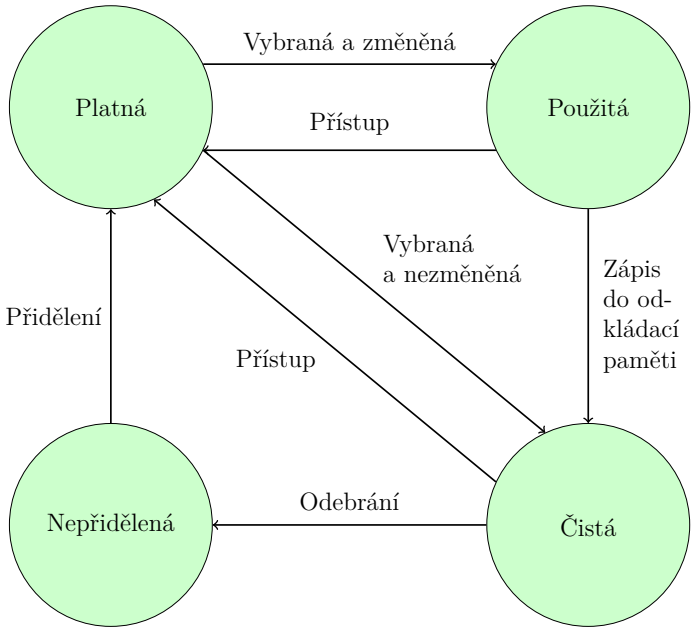
5.4 Stavy a sdílení stránek

Podobně jako procesy i stránky procházejí různými stavy. Její stav určuje následující akce se stránkou. Na obrázku 5.15 je zobrazen jednoduchý model stavů stránek [31]. Možné jsou tyto stavy:

- Nepřidělená (free) – stránku je možné alokovat pro proces.
- Platná (active) – stránka je přidělena procesu a je jím aktivně používána.
- Použitá (inactive dirty) – proces stránku nepoužívá, stránka byla vybrána k odstranění z hlavní paměti, byl změněn její obsah.
- Čistá (inactive clean) – proces stránku nepoužívá, stránka byla vybrána k odstranění z hlavní paměti, její obsah je stejný jako v odkládací paměti.

Všechny stránky dostupné pro přidělení procesům (alokování) se nachází ve stavu „volná“. Pokud je stránka přidělena procesu a je procesem aktivně využívána, je její obsah přesunut do hlavní paměti – stránka je ve stavu „platná“.

Stránky v hlavní paměti jsou periodicky prohledávány zvoleným algoritmem (LRU, NFU, Aging) a jsou vybrány stránky k odložení pro uvolnění rámců hlavní paměti. Při vybrání stránky následují dvě možnosti:



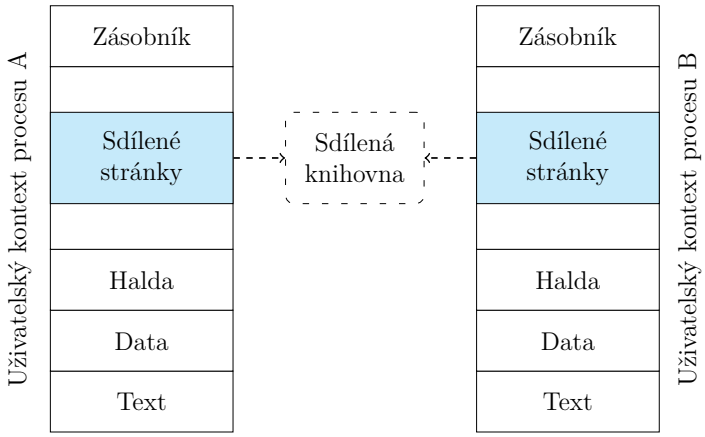
Obrázek 5.15: Jednoduchý model stránek.

- Vybraná stránka přechází do stavu „použitá“, pokud její obsah v hlavní paměti byl změněn oproti jejímu obsahu v odkládací paměti. Do stavu „čistá“ se stránka dostane zápisem (synchronizací) jejího obsahu do odkládací paměti.
- Pokud obsah v hlavní paměti změněn nebyl (data ve stránce jsou například pouze pro čtení), tak stránka přechází ihned do stavu „čistá“.

Stránka přechází do výchozího stavu „volná“ když proces stránku dále nepotřebuje (uvolnění paměti, ukončení celého procesu). Stránka může být přidělena jinému procesu.

Stavy stránek umožňují práci se sdílenými stránkami. Při spuštění stejného programu vícekrát je efektivní sdílet stránky pro vzniklé procesy. Stránky může také sdílet proces rodiče a potomka. Rodič i potomek na začátku (po vytvoření potomka) sdílí stránky s programovým kódem i stránky s měnícími se daty – „potomek je v paměti rodič“. Dokud procesy data pouze čtou, tak je situace neměnná. Jestliže jeden z procesů zapisuje data do sdílené stránky, je vytvořena kopie této stránky a zápis proveden zde. Výhodou je, že není automaticky vytvářena kopie všech stránek rodiče při vzniku potomka. Tím dochází k úspoře paměti.

Sdílet lze také stránky pro pouze čtení, např. data knihovny, viz obrázek 5.16 [16]. Stránky sdílené knihovny jsou umístěny v části uživatelského kontextu pro paměťově mapované soubory.



Obrázek 5.16: Umístění sdílené knihovny v paměťovém prostoru procesů v podobě sdílených stránek.

Při sdílení stránek mohou nastat tyto příkladové situace [11]:

1. Sdílená stránka je odložena. Jiný proces, který tuto stránku sdílí, ji vzápětí potřebuje – nastane zbytečný výpadek stránky (přesun stránky do hlavní paměti).
2. Proces ukončí svou činnost – je nutno ohlídat, které stránky končícího procesu jsou sdíleny s jinými procesy, aby nebyly uvolněny (označeny jako „nepřidělená“).

Prohledávání všech stránek a zjišťování, kterými procesy jsou využívány, je náročná operace. Operační systémy používají seznam sdílených stránek pro řešení těchto stavů.

5.5 Příklady na paměť

První příklad pracuje s odkládacím prostorem na úložném zařízení, který slouží pro realizaci virtuální paměti. Na příkladu je popsáno, kde se odkládací prostor nachází a jak ho dočasně navýšit. Druhý příklad se zabývá výpadky stránek se zobrazením jejich počtu v intervalu jedné sekundy. V posledním příkladu je demonstrováno použití anonymních stránek a jejich rozdílu oproti stránkám, které jsou svázané se souborem v souborovém systému. Zde je také předvedena ukázka sdílení stránek.

5.5.1 Odkládací prostor

Pro realizaci virtuální paměti se používá odkládací prostor na úložném zařízení, kam jsou přesunovány bloky paměti v podobě stránek z hlavní paměti. V operačním systému Linux je tento prostor realizován pomocí diskového oddílu. V OS Windows je odkládacím prostorem soubor. Který oddíl je využit jako odkládací lze zjistit příkazem `fdisk`[11], jak je zobrazeno ve výpisu 5.3. Tento oddíl je formátován na souborový systém, který je označen jako *Linux swap*.

```
[]# fdisk -l
Zarizeni    Id  System
```

[11] Je nutno právo administrátora

```
/dev/sda1   83   Linux
/dev/sda2   82   Linux swap
/dev/sda3   83   Linux
```

Výpis kódu 5.3: Odkládací prostor.

Aktuální stav virtuální paměti lze získat příkazem `free`, viz výpis 5.4. Z výpisu je patrné, že odkládací prostor je využit minimálně, tj. hlavní paměti je pro spuštěné programy dostatek.

```
[]$ free -h
total    used    free
Mem:    7,6G    2,8G    3,7G
Swap:   7,7G    767M    7,0G
```

Výpis kódu 5.4: Přehled využití paměti.

Jako odkládací prostor lze využít i soubor v souborovém systému. Uveďme si příklad pro PC s 8 GB RAM a odkládací oddíl 8 GB. Potřebujeme zpracovat ve vlastním programu velký textový soubor (nahrát celý do paměti, paralelní programování s kopiemi dat). Zpracování souboru ve špičce zabere 24 GiB. V současné konfiguraci se program předčasně ukončí z důvodu nedostatku paměti a úlohu nelze provést. Řešením je dočasně navýšit virtuální paměť bez nutnosti změny velikosti oddílů na paměťovém úložišti. Postup realizace je následující 5.5:

1. Vytvoření souboru o potřebné velikosti, například pomocí jeho naplnění nulovými daty.
2. Formátování souboru jako odkládacího.
3. Zavedení souboru do systému virtuální paměti.

Nejdříve je vytvořen soubor o velikosti 16 GiB s nulovými daty, která byla vygenerována pomocí softwarového zařízení `/dev/zero`. Dále byl soubor formátován pomocí příkazu `mkswap`. V posledním kroku byl soubor určen k použití jako odkládací (příkaz proveden jako *root*). Postup zahrnuje i úpravu vlastníka souboru a práv, tyto úkony jsou z důvodu jednoduchosti vynechány.

```
[]$ dd if=/dev/zero of=~/vmsoubor bs=1GiB count=16
17 179 869 184 bajtu (17 GB) zkopirovano
[]$ mkswap vmsoubor
Vytvarim odkladaci prostor velikost = 16777212 KiB
[]# swapon /home/komosny/vmsoubor
```

Výpis kódu 5.5: Navýšení velikosti odkládacího prostoru.

Ověření navýšení odkládacího prostoru (nyní oddíl i soubor) lze zjistit ve výpisu 5.6. Položka *PRIO* udává pořadí využití odkládacího místa; vyšší číslo značí vyšší prioritu. Tato konfigurace je platná do vypnutí systému. Pro trvalé využití odkládacího souboru je potřeba provést jeho automatické zavedení do virtuální paměti při startu systému.

```
[]$ swapon
NAME                    TYPE        SIZE  USED    PRIO
```

```
/dev/dm-0               partition   7,8G   744,8M   -2
/home/komosny/vmsoubor  file        16G    0B       -3
```

Výpis kódu 5.6: Ověření zavedení odkládacího souboru.

5.5.2 Výpadky stránek

K výpadku stránky dojde, pokud proces vyžaduje data ze stránky, která se aktuálně nenachází v hlavní paměti. Systém musí stránku vyhledat v odkládací paměti a přenést ji do hlavní paměti. Pokud v hlavní paměti není místo pro uložení nové stránky, systém vybere stránku a tu z hlavní paměti přesune do paměti odkládací. Operační systém může periodicky prohledávat stránky hlavní paměti a přesunovat je do odkládací paměti. Tímto je stále udržován určitý počet rámců v hlavní paměti volný pro rychlejší provedení výpadků stránek.

Počet výpadků stránek ovlivňuje výkonnost operačního systému. Větší frekvence výpadků stránek značí nedostatek hlavní paměti nebo malý počet rámců přidělených danému procesu. Dochází k značným přesunům dat mezi hlavní pamětí a odkládacím prostorem. Informace o počtu výpadků stránek v systému jsou zobrazeny na výpisu 5.7, který byl získán příkazem `sar` (interval aktualizace 1 s).

```
[]$ sar -B 1
14:36:57      fault/s   majflt/s
14:37:03      20,60     0,00
14:37:05      18,18     0,00
14:37:07      25,13     0,00
Average:      22,13     0,00
```

Výpis kódu 5.7: Výpadky stránek.

- Položka *fault/s* udává počet všech výpadků stránek za sekundu. Jedná se i o stránky, které negenerují I/O operace. Nepřítomnost I/O operací značí, že stránky jsou uloženy ve vyrovnávací paměti nad blokovým zařízením (úložným zařízením, viz blíže popis souborových systémů).
- Položka *majflt/s* udává počet výpadků stránek; stránky jsou přesunuty z odkládacího prostoru, tj. generují I/O operaci nad blokovým zařízením.

Jak je z výpisu vidět, v systému je dostatek hlavní paměti pro spuštěné procesy (nedochází k výrazným přesunům stránek, mimo vyrovnávací paměť nad blokovým zařízením).

5.5.3 Rozložení paměťového prostoru

Pro demonstraci rozložení paměťového prostoru bude využit program ve výpisu 5.8, který má název `pamet`. V programu je otevřen soubor `data.dat` v režimu pouze čtení. Následně je program zablokován pomocí nekonečné smyčky *while*.

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/mman.h>
```

```
4
5  int main()
6  {
7   int soubor = open("data.dat", O_RDONLY | O_CREAT);
8   while(1);
9  }
```

Výpis kódu 5.8: Program pro demonstraci rozložení paměťového obrazu procesu.

Rozložení paměťového prostoru procesu je zobrazeno ve výpisu 5.9. Povšimněte si, že otevřený soubor `data.dat` není součástí paměťového prostoru procesu (není vidět ve výpisu).

```
1  []$ pmap -x PID_procesu
2  Address    Kbytes    RSS    Dirty    Mode     Mapping
3  0400            4      4        0    r-x--    pamet
4  0601            4      4        4    rw---    pamet
5  ...
6  48ca           20     12       12    rw---    [ anon ]
7  48cf          136    108        0    r-x--    ld-2.17.so
8  4acf           12     12       12    rw---    [ anon ]
9  4aef            4      4        4    rw---    [ anon ]
10 ...
11 433f          132     16       16    rw---    [ stack ]
```

Výpis kódu 5.9: Rozložení paměťového prostoru procesu.

Položka *Dirty* udává, u kolika stránek byl změněn jejich obsah oproti odkládacímu prostoru. Údaj je uveden v KiB a velikost jedné stránky v použitém systému je 4 KiB. Připomeňme si, že v případě jejich výběru pro odstranění z hlavní paměti musí být provedené změny přeneseny do odkládacího prostoru, viz popis stavů stránek na obrázku 5.15.

Sloupec *Mapping* zobrazuje popisný název pro každou oblast. Oblasti zahrnující programový kód a inicializovaná data jsou v paměťovém obrazu označeny jako *pamet* (název spuštěného programu). Oblasti se odlišují různými právy pro přístup. Pokud je zde uvedeno právo spustitelnosti x, tak se jedná o programový kód procesu. Pokud je u oblasti uveden příznak možnosti zápisu (w), tak se jedná o měnitelná data. Blíže viz rozložení uživatelského kontextu v kapitole 4.2.1. Na konci výpisu je uveden zásobník (stack).

5.5.4 Anonymní stránky

Anonymní stránka je taková, která není svázaná se souborem v souborovém systému. U anonymních stránek se pro odkládání jejich obsahu používá odkládací prostor (diskový oddíl nebo vyhrazený soubor).

Část paměti procesu, která je svázaná se souborem (paměťově mapovaný soubor), je ve výpisu 5.9 pojmenována jako tento soubor, např. dynamický linker (dynamic linker) `ld.so` – jedná se o zavaděč programu, který načte použité sdílené knihovny do adresního prostoru procesu a následně je sváže s programem. U ostatních částí paměti procesu (nemají odpovídající soubor) je zobrazen popisek `[ anon ]`, tato oblast je tvořena anonymními stránkami.

Abychom zobrazili rozdíl mezi anonymními stránkami a stránkami, které jsou svázané se souborem v souborovém systému, provedeme následující jednoduchou úpravu programu, viz výpis 5.10. Pomocí funkce *mmap* byl namapován soubor `data.dat` do uživatelského kontextu procesu. Použité stránky mají odpovídající soubor v souborovém systému (což je v kontrastu s anonymními stránkami, které odpovídající soubor nemají). Připomeňme si, že pokud je potřeba odstranit anonymní stránku z hlavní paměti, tak je přesunuta do odkládacího prostoru.

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/mman.h>

int main()
{
 int soubor = open("data.dat", O_RDONLY | O_CREAT);
 mmap(0,1,PROT_READ,MAP_PRIVATE,soubor,0);
 while(1);
}
```

Výpis kódu 5.10: Program pro paměťově mapovaný soubor.

Vybrané parametry použité funkce *mmap* (důležité pro demonstraci) jsou tyto:

- 1. parametr – adresa stránky, od které se má začít mapovat (0 - jádro zvolí adresu první stránky automaticky),
- 2. parametr – počet bytů k namapování, zde je uveden 1 B,
- 5. parametr – deskriptor souboru pro namapování.

Po spuštění programu se paměťový obraz změní do podoby zobrazené ve výpisu 5.11. Do paměťového prostoru procesu byla přidána oblast o velikosti 4 KiB, která je použita pro namapovaný soubor `data.dat`. Povšimněte si, že namapovány byly 4 KiB místo požadovaného 1 B, protože velikost stránky v systému je 4 KiB. Připomeňme si, že při práci s paměťově mapovanými soubory se používá jako jednotka dat jedna stránka – tedy 4 KiB. Dochází k plýtvání paměti, jelikož zbytek stránky nelze použít pro jiná data. Dále si povšimněte, že hodnota *Dirty* je u této stránky 0. Důvod je ten, že soubor byl otevřený pouze pro čtení, a tak nemůže dojít ke změně dat ve stránce oproti jejímu obsahu v odkládacím prostoru (zde souboru).

```
Address     Kbytes    RSS    Dirty    Mode     Mapping
0400             4      4        0    r-x--    pamet
0601             4      4        4    rw---    pamet
...
feb5            12     12       12    rw---    [ anon ]
fed4             4      0        0    r----    data.dat
fed5             4      4        4    rw---    [ anon ]
...
1336           132     12       12    rw---    [ stack ]
```

Výpis kódu 5.11: Rozložení paměťového obrazu procesu s paměťově mapovaným souborem.

Stránky lze mezi procesy sdílet. Toto sdílení šetří paměť a slouží pro komunikaci mezi procesy pomocí zápisu a čtení z této paměti. Zdrojový kód drobně upravíme tak, že paměťově mapovaný soubor bude sdílený. To dosáhneme tak, že při mapování souboru předáme příznak sdílení (MAP_PRIVATE je nahrazeno za MAP_SHARED), jak je uvedeno ve výpisu 5.12.

```
int main()
{
 int soubor = open("data.dat", O_RDONLY | O_CREAT);
 mmap(0,1,PROT_READ,map_shared,soubor,0);
 while(1);
}
```

Výpis kódu 5.12: Program pro paměťově mapovaný soubor, který je sdílený.

Po kompilaci a spuštění programu lze pozorovat rozdíl v příznacích stránky, která obsahuje tento mapovaný soubor. Nyní se jedná o sdílenou stránku (příznak `s`), viz výpis 5.13.

```
Address   Kbytes  Dirty  Mode   Mapping
ddb76000  12      12     rw---  [ anon ]
ddb95000  4       0      r--s-  data.dat
ddb96000  4       4      rw---  [ anon ]
```

Výpis kódu 5.13: Rozložení paměťového obrazu procesu s paměťově mapovaným souborem, který je sdílený.

6 SOUBOROVÉ SYSTÉMY

Souborový systém definuje, jak jsou organizována data na paměťovém médiu, kterým může být například pevný disk nebo paměť typu flash[1]. Paměťové médium může být fyzicky umístěné přímo ve výpočetním prostředku, nebo může být přístupné vzdáleně pomocí sítě. Síťový souborový systém je takový, jehož data jsou dostupná pomocí počítačové sítě. Data jsou organizována ve formě souborů. Soubory bývají pro snadnější orientaci sdružovány do adresářů. Plochý (flat, single-level) souborový systém je takový, který má pouze jeden adresář a všechny soubory jsou umístěny v tomto adresáři. Tento systém je používán pro menší počty souborů a v jednouživatelských systémech [16].

Souborové systémy udržují informace o **umístění** dat jednotlivých souborů na paměťovém médiu. Mimo tuto základní informaci mohou také poskytovat **časové informace**, například čas vytvoření a poslední modifikace souboru. Ve víceuživatelských systémech poskytují také informace o **vlastnících** a přístupových **právech**, například právo čtení/-zápisu/spouštění pro vlastníka nebo skupinu uživatelů.

Informace souborových systémů jsou děleny na data uložená v souborech a **metadata** [16]. Pod pojmem data jsou zahrnuty vlastní informace souborů (jejich obsah). Metadata jsou data potřebná pro práci se soubory, tedy například již výše uvedené informace o umístění dat souboru na paměťovém médiu, časové informace nebo přístupová práva.

Paměťová média se typicky rozdělují na několik samostatných částí viz obrázek 6.1. **MBR** (Master Boot Record) je typ spouštěcího záznamu v prvním sektoru (viz dále) paměťového média. MBR obsahuje zavaděč operačního systému, identifikátor disku a **tabulku oddílů** (partition table)[2]. Zavaděč v MBR je spouštěn programovým kódem uloženým v BIOS (Basic Input-Output System). Úkolem zavaděče je načíst zaváděcí (boot) sektor z aktivního oddílu (jeden oddíl v tabulce označen jako aktivní). Aktivní oddíl obsahuje operační systém, který se následně spustí. Zaváděcí sektor je individuální pro daný operační systém. V tabulce oddílů jsou uloženy údaje o začátku a konci jednotlivých oddílů (partition) na paměťovém médiu. Na oddílech se nachází souborové systémy. Na různých oddílech mohou být různé souborové systémy a na těch pak mohou být dostupné různé operační systémy.

Obrázek 6.1: Dělení paměťového média.

6.1 Datové bloky a metadata

Souborové systémy ukládají soubory do **datových bloků** (fragmentů). Bloky o větší velikosti znamenají, že pro malý soubor musí být použit celý blok a v tomto bloku pak

[1]Paměťovým médiem může být například i paměť RAM, kde může být umístěn dočasný souborový systém. Například se může jednat o data potřebná při startu OS, kdy ještě nejsou dostupná data z pevného disku.

[2]Volitelně může obsahovat i další části.

může zbýt volný prostor. Protože jeden blok je základní jednotka při ukládání souborů, volný prostor v tomto bloku nelze použít pro data jiného souboru a vzniká tak **interní fragmentace**. Na druhé straně použití malých bloků značí, že pro jeden soubor může být použito velké množství datových bloků. Práce s každým blokem vyžaduje vyhledání jeho umístění na paměťovém médiu, což může být časově náročná operace. Malá velikost datových bloků tedy přináší zpomalení operací se soubory.

Souborové systémy mohou používat různé velikosti datových bloků. Jejich velikost lze ovlivnit při formátování (vytváření) souborového systému na oddílu paměťového média.

6.1.1 Ukládání datových bloků

Pro ukládání souborů v souborovém systému existují tři základní způsoby [11, 30]:

- bloky souboru mohou být uloženy po sobě – kontinuálně (contiguous allocation),
- bloky souboru jsou rozloženy nesouvisle; datové bloky navíc obsahují ukazatele, které je navzájem svazují (linked allocation),
- bloky souboru jsou rozloženy nesouvisle; ukazatele na datové bloky jsou umístěny zvlášť v indexačním bloku (indexed allocation).

Ukládání bloků kontinuálně

V prvním případě jsou bloky souboru uloženy na paměťovém médiu za sebou. Tento způsob ukládání je vhodný u plotnových disků, jelikož načtení bloků celého souboru je rychlé, protože nevyžaduje přesun čtecích hlav (mimo přesuny hlav mezi cylindry, viz dále). Adresářový záznam pro takto uložené soubory obsahuje adresu počátku dat souboru a jeho velikost (v podobě násobku počtu bloků). Kontinuální ukládání bloků ovšem neumožňuje jednoduchou expanzi nebo smršťování souborů. Zásadní je problém s konstantně rostoucím souborem (udržujícím například log aplikace nebo systému).

Jak jsou soubory vytvářeny, mazány a je měněna jejich velikost, dochází k dělení volného prostoru na paměťovém médiu na nesouvislé části, dochází tak k **externí fragmentaci**. Problém nastane, když největší souvislá část volného prostoru není dostatečná pro vykonání určité operace se souborem, i když celková velikost volného prostoru je dostačující. V dřívějších dobách se problém fragmentace volného prostoru řešil tak, že všechna data byla naráz zkopírována z jednoho souborového systému (celého paměťového média nebo jeho oddílu) na jiný souborový systém (v jiném oddílu nebo na paměťovém médiu). Původní data v souborovém systému byla smazána a byl tak sjednocen volný prostor – byla provedena jeho defragmentace. Následně byla data opět zkopírována na původní souborový systém a byla tak uložena kontinuálně. Tato operace byla časově náročná a bylo ji nutno provádět v pravidelných intervalech.

Ukládání bloků nesouvisle – ukazatele v datovém bloku

V případě rozložení bloků na paměťovém médiu nesouvisle obsahuje adresářový záznam o souboru ukazatel na první a poslední jeho datový blok. Každý blok souboru obsahuje ukazatel na následující blok téhož souboru. Velikost bloku dostupná pro ukládaná data je

tedy zmenšena o velikost ukazatele v něm obsaženém. Při vytváření souboru je nalezen volný blok kdekoliv na paměťovém médiu a tento blok je použit pro nová data. Pokud má soubor nulovou velikost, je ukazatel v adresáři nastaven jako neplatný. S prvním blokem souboru je tento ukazatel aktualizován. U rozšiřování souboru je v předchozím bloku vytvořen odkaz na nový blok. Při této strategii nenastává výše popsaný problém externí fragmentace – každý volný blok může být využit pro data souboru.

Problémem tohoto způsobu je nesouvislé čtení/zápis dat ze souboru. Pro načtení určitého bloku souboru (například chceme načíst data uprostřed souboru), je nutno začít načtením prvního bloku, zjistit ukazatel na následující blok, načíst tento následující blok atd. Každé načtení bloku, které mohou být rozmístěny kdekoliv na paměťovém médiu, značí časové zdržení (rotační disky). Další nevýhoda, která již byla zmíněna, je velikost ukazatele. Pokud je například velikost ukazatele 4 B a velikost datového bloku 512 B, tak 0,78 % každého bloku nemůže být využito pro vlastní ukládaná data. Částečným řešením těchto nevýhod je sdružit určitý počet bloku do skupin. Ukazatele na skupiny bloků zabírají méně procent velikosti celé skupiny. Zvyšuje se také rychlost práce s daty (čtení/-zápis). Nevýhodou je interní fragmentace, která nastává, když ukládaná data ve skupině bloků mají menší velikost, než je velikost této skupiny. Nevyužitý prostor v této skupině bloků nelze použít pro data jiného souboru. Jako další problém zmiňme možnost poškození ukazatele na další datový blok (například závada paměťového média). Od poškozeného ukazatele dále dojde ke ztrátě dat, tj. obsahu následujících bloků.

Ukládání bloků nesouvisle – ukazatele v indexačním bloku

Tento způsob ukládá ukazatele na datové bloky souboru na zvláštním místě – v indexačním bloku. Každý soubor má svůj vlastní indexační blok, který obsahuje seznam ukazatelů na bloky souboru s vlastními daty. Adresář se souborem v tomto případě obsahuje ukazatel na indexační blok tohoto souboru. Toto řešení umožňuje přímé načtení zvoleného datového bloku (například blok obsahující prostředek dat souboru). Při vytvoření souboru jsou všechny ukazatele v indexačním bloku neplatné. Při zápisu dat do volného datového bloku je provedena aktualizace ukazatele na tento blok.

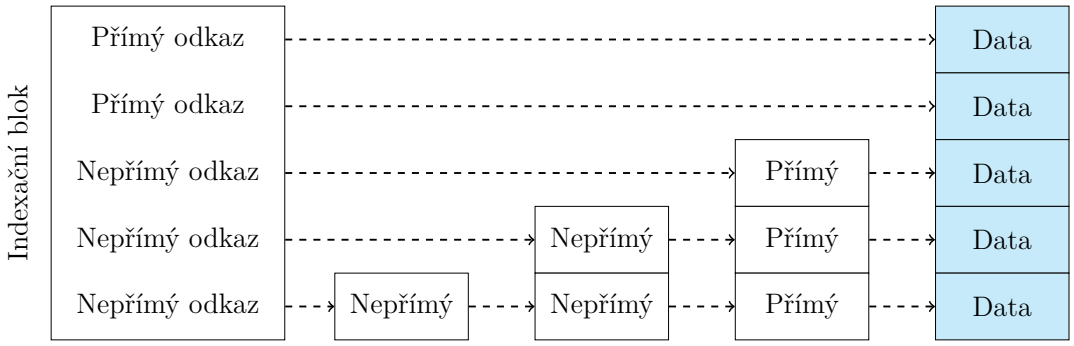
Použití indexačního bloku nezpůsobuje externí fragmentaci. Ovšem nevýhoda je v zabírání více místa pro ukazatele na datové bloky. Příkladem může být malý soubor o velikosti dvou bloků. Zde bude zabrán celý datový blok (například o velikosti 512 B), který je vyhrazený pro uložení indexačního bloku a to pouze pro dva ukazatele, například o velikosti 4 B. Důležitá je tedy i vhodná velikost indexačního bloku podle předpokládané velikosti souborů a datových bloků.

Používají se tyto přístupy pro práci s indexačními bloky:

- Indexační blok obsahuje ukazatele na datové bloky – **přímé odkazy**. V případě velkého souboru lze propojit více indexačních bloků. Poslední ukazatel je nastaven buď jako neplatný (malý soubor) nebo ukazuje na další indexační blok s ukazateli na datové bloky (velký soubor).
- Indexační blok obsahuje ukazatele na další indexační bloky – **nepřímé odkazy**. Tyto další indexační bloky pak obsahují ukazatele na datové bloky souboru (přímé odkazy). Vzniká hierarchické uspořádaní s několika úrovněmi. Pro přístup k datům určitého bloku se nejdříve načte indexační blok na první úrovni, získá se nepřímý

odkaz na indexační blok na druhé úrovni, který se načte, a v tomto indexačním bloku se získá přímý odkaz na požadovaný datový blok.

- Je také možná kombinace obou přístupů, viz obrázek 6.2 [11]. Indexační blok v tomto případě obsahuje přímé i nepřímé odkazy. Výhoda je v tom, že malé soubory lze pokrýt jedním indexačním blokem pomocí přímých odkazů. Pro velké soubory je pak využito nepřímých odkazů s možnými několika úrovněmi odkazování.



Obrázek 6.2: Příklad kombinace přímých a nepřímých odkazů na datové bloky.

Uveďme si příklad k poslednímu způsobu s tímto předpokladem – datový blok v souborovém systému má velikost 1 KiB a je adresovatelný ukazatelem o velikosti 4 B. Jeden blok může pak obsahovat až 256 ukazatelů na jiné bloky. Při použití 10 přímých odkazů, 1 bloku pro nepřímou indexaci první úrovně, 1 bloku pro nepřímou indexaci do druhé úrovně a 1 bloku pro nepřímou indexaci do třetí úrovně je maximální velikost adresovatelného souboru 16 GiB podle výpočtu $10 \times 1\,\text{KiB} + 256 \times 1\,\text{KiB} + 256 \times 256 \times 1\,\text{KiB} + 256 \times 256 \times 256 \times 1\,\text{KiB} = 16\,\text{GiB}$.

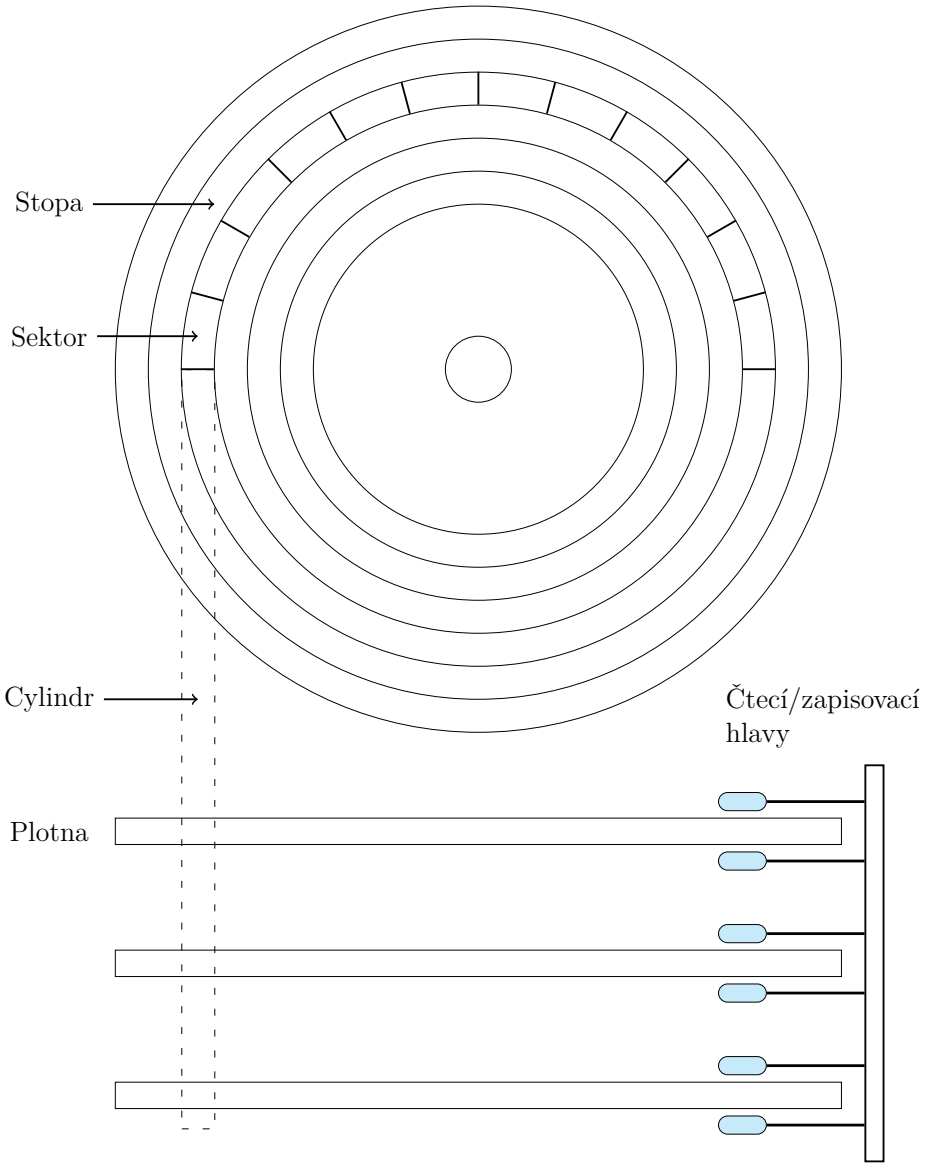
Nevýhoda ukládání ukazatelů do indexačního bloku je v nutnosti načítání dalších bloků pro přístup k datovým blokům (situace podobná pro ukládání bloků kontinuálně).

6.1.2 Ukládání dat a metadat

Data jsou ukládána na paměťové médium v podobě sektorů. Sektory rozlišujeme fyzické a logické. Fyzický sektor může mít stejnou nebo větší velikost než logický. Velikost fyzického sektoru udává nedělitelnou jednotku dat, která se používá pro čtení a zápis na médium. Naproti tomu logická velikost sektoru je objem dat, která lze načítat a ukládat na médium. Při nesouladu velikostí (logický sektor je menší než fyzický) se pro načtení logického sektoru ve skutečnosti načte celý fyzický sektor a je použita pouze jeho logická část. Obdobně, při zápisu logického sektoru je potřeba nejdříve načíst příslušný fyzický sektor. Následně aktualizovat jeho logický sektor a pak uložit celý fyzický sektor. Datové bloky definované souborovým systémem mohou mít velikost jednoho nebo více sektorů.

Problémem pevných disků realizovaných pomocí ploten je vzdálenost mezi daty a metadaty (informace o uložení dat souboru, přístupová práva atd.). Na obrázku 6.3 je zobrazeno rozložení povrchu ploten. Povrch plotny je rozdělen na **stopy** a ty jsou dále rozděleny na **sektory**. Plotny rotačního disku jsou uspořádány nad sebou. Pro čtení z povrchů ploten

se používá více hlav. Při práci s daty je potřeba nejdříve načíst metadata a z nich vyčíst informace o uložení vlastních dat (případně další informace o přístupových právech). Tato akce obnáší nejprve nastavení hlav na pozici, kde jsou datové bloky s metadaty a až pak následně na pozici, kde jsou datové bloky s vlastními daty souboru. Přesun hlav mezi metadaty a daty přináší časová zdržení. Z tohoto důvodu jsou plotnové disky dále rozděleny na několik „samosprávných“ oblastí, které obsahují data souboru a k nim příslušná metadata. Tyto oblasti se nazývají **cylindrické skupiny**, viz obrázek 6.3. Cylindr je sada stop, které jsou umístěny nad sebou na plotnách disku. Pokud je hlava jednoho povrchu nad stopou, ostatní hlavy jsou nad ostatními stopami z jiných povrchů a tyto stopy vytváří cylindr. Cylindrická skupina zahrnuje několik cylindrů a společně obsahuje metadata i data. Nedochází tak k nadměrným přesunům hlav při práci s metadaty a vlastními daty souborů [11].



Obrázek 6.3: Cylindr pevného disku.

Při popisu způsobu ukládání datových bloků spojitě byl zmíněn problém externí fragmentace, který vzniká tak, že volné datové bloky nejsou umístěny kontinuálně za sebou. Problém externí fragmentace je řešen pomocí ukazatelů na datové bloky souboru, které nemusí být uloženy souvisle – vzniká **fragmentace obsahu** souborů. U plotnových disků je práce s datovými bloky souboru (čtení/zápis) uloženými souvisle rychlejší, než pokud jsou bloky souboru rozloženy na jiných místech. Důvodem je pohyb hlav mezi různými stopami plotny. Rychlost práce s daty je také dána rychlostí otáčení ploten disku. Po přesunu hlavy na určitou stopu musí ještě plotna dorotovat na daný sektor. Pro zvýšení rychlosti práce s daty se provádí defragmentace obsahu souborů, tj. spojení datových bloků souboru do jednoho navazujícího celku.

U paměťových médií typu SSD (Solid State Drive), kde nejsou pohyblivé části, tato situace nenastává. Z toho důvodu se zde defragmentace neprovádí. Dalším důvodem je skutečnost, že počet zápisů u těchto médií je technologicky limitován. Překročení tohoto limitu může způsobit nefunkčnost paměťového média. Defragmentace je zde tedy z pohledu možností paměťového média naopak nežádoucí, jelikož navyšuje počet zápisů. Naopak ovladač paměťového média může provádět zápis dat tak, aby celkový počet zápisů byl rozložen rovnoměrně nad celým médiem a nedocházelo tak opakovaným zápisům pouze nad určitou oblastí.

6.1.3 Konzistence dat a metadat

Při práci s daty v souborových systémech (zápis, mazání) může nastat situace, že dojde k přerušení dané činnosti z důvodu pádu systému (výpadek napájení, systémový pád). To následně může vést k zanechání dat v **nekonzistentním stavu**. Situace, které mohou například nastat:

- Soubor nebyl smazán, i když byl zadán požadavek k jeho smazání.
- Část dat byla aktualizována zápisem, další část již ne.

Rozeberme si blíže operaci mazání souboru, která zahrnuje tři dílčí kroky:

1. odstranění záznamu o souboru z adresáře,
2. uložení čísla i-uzlu[3] souboru do seznamu volných i-uzlů,
3. uložení datových bloků souboru do seznamu volných bloků k použití.

Pokud pád systému nastane mezi krokem 1 a 2, tak v souborovém systému zůstane i-uzel ve stavu použitém, i když použitý není. I-uzel zabírá určité místo v souborovém systému, a to nemůže být použito pro jiná data. Podobná situace může nastat při pádu systému mezi krokem 2 a 3 – ovšem pro datové bloky souboru a z pohledu volného místa v souborovém systému je tato situace závažnější.

Pro předcházení vzniku nekonzistentních dat v souborovém systému se používají **žurnálovací souborové systémy** (journal file systems) [32, 16]. Tyto souborové systémy udržují ve svém „žurnálu“ plánované změny dat před jejich vlastním provedením. V případě pádu systému je provedena obnova dat pomocí načtení předešle uloženého plánu operací v žurnálu. Uložené operace jsou provedeny znovu do té fáze, kdy jsou data a

[3]Jedná se o datovou strukturu obsahující metadata. Každý soubor má svůj i-uzel, ve kterém jsou uložena metadata k tomuto souboru. I-uzel bude probrán podrobněji v kapitole 6.3.3.

metadata v souborovém systému konzistentní. Každá plánovaná operace v žurnálu je proveditelná pouze jako jeden celek, tj. provedená kompletně před pádem systému, nebo znovu provedená kompletně po pádu systému. Pokud není operace při pádu systému jako celek zapsána do žurnálu, tak je ignorována. Kontrola, že zápis dat do žurnálu nebyl přerušen, může být provedena pomocí kontrolního součtu údajů v žurnálu. V případě pádu a nedokončeného zápisu dat do žurnálu, nebude kontrolní součet odpovídat a nedokončený zápis dat v žurnálu bude ignorován. Plánované operace v žurnálu mohou být udržovány v podobě cyklického logu s danou velikostí. V případě úspěšné operace je do žurnálu přidána informace o jejím provedení a poté může být záznam v žurnálu smazán.

Do žurnálu se pro plánované operace mohou ukládat pouze metadata, nebo metadata i vlastní data souborů. První varianta přináší rychlejší žurnálovací operace za cenu možné ztráty konzistence dat. Druhá varianta značí pomalejší žurnálovací operace, ale garantuje plné provedení operací se soubory.

- Pokud se do žurnálu ukládají pouze metadata, může nastat situace inkonzistence mezi metadaty. Příkladem může být přidání dat k existujícím datům v souboru. Nejprve je změněn záznam o velikosti souboru v i-uzlu daného souboru (metadata). Dále je z volných datových bloků vybrán prostor pro nová data (metadata). Posledním krokem je, že do vybraných volných datových bloků jsou zapsána přidaná data souboru (data). Při ukládání pouze metadat do žurnálu nebude pokryta poslední operace zápisu dat do datových bloků. V případě pádu bude tedy soubor rozšířen o nová data, kterými budou předchozí data v datových blocích (náhodná data, a tedy ne data určená pro přidání k souboru).
- Při práci s vlastními daty souborů jsou do žurnálu ukládány kopie všech datových bloků, které se následně budou zapisovat na paměťové médium. V případě pádu systému při zápisu, je zápis datového bloku proveden znovu pomocí kopie datového bloku uloženého v žurnálu. Z důvodu vytváření kopií všech ukládaných dat je zápis značně zpomalován. Tento režim práce žurnálu se používá v systémech, kde je vyžadována kompletní ochrana dat při pádu systému.

Provedení nedokončených změn zaznamenaných v žurnálu je provedeno při následujícím připojení souborového systému. Implementace žurnálovacího souborového systému má několik základních variant:

1. Kopie žurnálu mohou být umístěny na různých paměťových mediích z důvodu jejich zálohy v případě poruchy jednoho média.
2. Plánované změny v žurnálu mohou být dále ukládány v jiném žurnálu z důvodu zvýšení redundance.
3. Žurnál může být ukládán na stejném paměťovém médiu, nebo na jiném z důvodu vyšší rychlosti druhého média.

6.1.4 Virtuální soubory

Mimo „reálné“ souborové systémy, které ukládají reálná data do souborů na typická paměťová média, se v operačních systémech používají i **virtuální souborové systémy** [16],

které udržují virtuální soubory[4]. Data se mohou ve virtuálních souborech generovat dynamicky teprve na základě požadavku, tj. před požadavkem v systému neexistují. Virtuální soubory se například používají pro komunikaci mezi programem jádra a uživatelskými procesy tak, že zobrazují informace z paměťové oblasti jádra v podobě běžných souborů[5].

Virtuální souborové systémy také mohou obsahovat soubory, které reprezentují zařízení. Takovéto virtuální – speciální – soubory umožňují přímý přístup k danému zařízení. Pomocí čtení a zápisu do souboru lze se zařízením pracovat. Pro práci se zařízeními jsou tedy použity stejné operace jako pro práci s obyčejnými soubory, což přináší abstrakci při jejich používání. Speciální soubory mohou reprezentovat fyzická a softwarová zařízení. Příkladem fyzických jsou úložná zařízení, sériové porty a zvukové karty. Softwarová zařízení mohou být generátory dat. V případě hardwarových zařízení se mohou soubory vytvářet automaticky a reflektovat tak změny v aktuálně dostupných zařízeních. Rozlišujeme dva druhy zařízení, která jsou reprezentována speciálními soubory – znaková (character) a bloková (block). Tato zařízení se liší jednotkou dat, se kterou pracují při zápisu či čtení.

- Znaková zařízení pracují s jednotkou jeden znak. Nepoužívají vyrovnávací paměť a umožňují tak přímý přístup k zařízení. Jedná se například o tiskárny a terminály. Nad znakovými zařízením nelze provozovat souborové systémy.
- Bloková zařízení pracují s jednotkou blok dat, který může mít různou velikost. Využívají vyrovnávací paměť, což vede k rychlejší práci se zařízením. Pokud je potřeba ze zařízení načíst menší data než jeden blok, je načten do vyrovnávací paměti celý blok, ze kterého jsou pak požadovaná data předána. Nad blokovými zařízením se provozují souborové systémy.

6.2 Standard pro organizaci souborů a adresářů

Operační systém může obsahovat velké množství souborů. Je tedy nutné jejich umístění organizovat. Existují různé varianty organizace souborů a adresářů. Tato organizace může být specifická pro typ (i verzi operačního systému), nebo může být stejná pro celou skupinu operačních systémů. Rozšířený je standard **FHS** (Filesystem Hierarchy Standard) [35]. Jeho znalost dává možnost „předvídat“, kde se který soubor či adresář nachází napříč různými operačními systémy, které standard přímo nebo jeho varianty používají. Základem standardu je dělení souborů podle těchto dvou zásad:

- Statické soubory (nemění svůj obsah) jsou odděleny od dynamických souborů (mění svůj obsah). Důvodem je možnost uložení statických souborů na paměťové médium s přístupem pouze pro čtení nebo připojení souborového systému pouze pro čtení. U dynamických souborů může být v pravidelných intervalech prováděna jejich záloha.

[4]Také označovány jako „pseudo-soubory“.

[5]Virtuální souborový systém `procfs` zobrazuje systémové informace a informace o procesech v adresáři `/proc`. Například v souboru `/proc/`<PID>`/cwd` je informace o aktuálním pracovním adresáři procesu s daným PID.

	Sdílitelné	Nesdílitelné
Statické	/usr,/opt	/boot,/etc
Dynamické	/var/mail	/var/lock,/var/log

Tabulka 6.1: Základní dělení adresářů pro jejich organizaci.

- Soubory specifické pro jednu stanici (nesdílitelné) jsou odděleny od souborů, které jsou použitelné pro více stanic (sdílitelné). Důvodem je jejich organizované sdílení po síti. Příkladem sdílených souborů jsou přenositelné skripty.

Příklad adresářů podle těchto zásad dělení je uveden v tabulce 6.1. V tabulce je u adresáře s dynamickými daty `/var` rozlišen podadresář `mail` (sdílitelná data), kde jsou uloženy mailové schránky uživatelů. Dále jsou v tabulce uvedeny nesdílitelné podadresáře `lock` a `log`. V adresáři `lock` jsou ukládány „zamykací“ soubory pro řízení přístupu k různými zdrojům mezi více procesy[6]. V adresáři `log` jsou soubory se záznamy činnosti systému a aplikací.

6.2.1 Primární organizace

Standard definuje povinné a volitelné adresáře na první úrovni v kořenovém adresáři `/`. Všechny soubory a adresáře spadají (přímo či nepřímo) pod tento adresář. Soubory mohou být umístěny na různých paměťových médiích a v různých souborových systémech, včetně virtuálních. Povinné adresáře na první úrovni (seznam není kompletní) jsou:

- `/bin` – Obsahuje binární spustitelné soubory pro základní práci s operačním systémem. Tyto soubory jsou určeny pro běžné uživatele i administrátory (viz rozdíl oproti `/sbin`). Standard určuje, které soubory musí adresář obsahovat. Například se jedná o binární soubory s programy realizující základní příkazy `cat`, `chmod`, `chown`, `kill` a `rm`.
- `/boot` – Zahrnuje soubory pro start operačního systému. Adresář obsahuje data pro počáteční práci operačního systému před spuštěním programů v uživatelském režimu. Adresář může obsahovat jádro operačního systému[7]. Příkladem je binární soubor s jádrem systému `vmlinuz` a obraz RAM disku `initramfs`.
- `/dev` – Obsahuje speciální soubory zařízení. Příkladem jsou speciální soubory reprezentující oddíly na disku `sda1`, `sda2` a soubory reprezentující softwarová zařízení `random`, `zero`.
- `/etc` – Zde jsou uloženy konfigurační soubory. Jedná se o statické soubory. Výjimkou je soubor `/etc/mtab`, který obsahuje dynamické informace o připojených souborových systémech (viz příklady). V tomto adresáři nesmí být umístěny spustitelné soubory. Příkladem je podadresář `httpd/` s konfiguračními soubory webového serveru a podadresáři `X11/` s konfiguračními soubory grafického rozhraní X window.

[6] Prezence zamykacího souboru značí aktuální používání prostředku.
[7] Další varianta pro umístění jádra je přímo v kořenovém adresáři.

- `/lib` – Zde jsou uloženy knihovny, které jsou potřeba pro start systému a činnost programů umístěních v adresářích `/bin` a `/sbin`. Obsahuje také podadresář `modules` s moduly jádra. Příkladem jsou soubory `ext4` a `fat` s moduly realizující ovladače souborových systémů.
- `/media` – Místo pro připojení přenosných úložných zařízení. Příkladem je soubor reprezentující připojený souborový systém na přenosném médiu (FLASH disk).
- `/mnt` – Místo pro dočasně připojované souborové systémy. Příkladem je soubor reprezentující dočasně připojený síťový souborový systém.
- `/opt` – Obsahuje instalované programy[8], každému typicky odpovídá vlastní podadresář.
- `/sbin` – Obsahuje systémové nástroje pro správu operačního systému. Jejich použití je typicky omezeno pro administrátory a jsou potřeba pro údržbu a konfiguraci systému. Příkladem jsou binární soubory s programy realizující příkazy pro správu uživatelů `useradd`, `usermod` a `userdel`.
- `/tmp` – Slouží k ukládání dočasných souborů. V některých systémech se tento adresář automaticky promazává, například při startu systému.
- `/usr` – Další hierarchie adresářů. Jedná se o sdílitelná data, která mohou být dostupná pouze ke čtení. Příkladem je adresář `src` se zdrojovými kódy programů. Bližší popis této sekundární organizace adresářů je uveden dále.
- `/var` – Obsahuje dynamická data. Zahrnuje například podadresáře s logy programů/celého systému. V případě provozovaného webového serveru jsou zde data webových stránek. Povinné podadresáře jsou například `lock` a `log`, viz jejich popis v tabulce 6.1. Příkladem je soubor `/var/log/messages` se záznamy o činnosti OS.

Volitelné adresáře na první úrovni jsou následující (výčet není kompletní):

- `/home` – Obsahuje domovské adresáře uživatelů nazvané většinou podle jména uživatelského účtu.
- `/root` – Domovský adresář uživatele *root*.

6.2.2 Sekundární organizace

Povinný adresář `/usr` tvoří druhou hierarchickou strukturu adresářů. Tento adresář obsahuje sdílitelná data pouze pro čtení. Nemají zde být umístěny soubory určené jen pro danou stanici nebo s měnícím se obsahem. Tímto je umožněno sdílení dat v tomto adresáři mezi různými systémy. Adresář také obsahuje povinné a volitelné podadresáře. Povinné podadresáře jsou:

- `bin` – Obsahuje binární soubory s programy realizující příkazy pro práci běžných (neprivilegovaných) uživatelů. Příkladem jsou soubory s programy `perl` a `python`.
- `lib` – Obsahuje knihovny. Příkladem je podadresář `gcc` s knihovnami pro kompilaci programů v jazyce C++.

[8]Instalované programy se umísťují na několik míst, toto je jedno z možných.

- `local` – Zde je možné instalovat programy.
- `sbin` – Obsahuje binární soubory s programy realizující příkazy určené pro správce systému, které nejsou zásadní pro operační systém (programy nutné pro správu systému jsou umístěny v adresáři `/sbin`). Příkladem jsou soubory s programy pro správu oddílů a souborových systémů `fdisk`, `mkfs` a `fsck` .
- `share` – Zde jsou umístěna data sdílitelná mezi dalšími systémy. Nachází se zde například manuálové stránky programů a příkazů dostupné pomocí příkazu `man`. Bývají zde umístěny také různé slovníky, například se slovy anglického jazyka[9].

Dále se v adresáři `/usr` mohou nacházet různé volitelné podadresáře. Příkladem může být adresář `include`, který obsahuje hlavičkové soubory pro programy v jazyce C, a `src`, který obsahuje zdrojové kódy programů.

6.3 Příklady na souborový systém

Příklady demonstrují získání informací o podporovaných souborových systémech (reálných i virtuálních) a dostupných úložných zařízeních. Je také ukázán princip metadat. Příklady jsou ukončeny využitím speciálních souborů pro generování náhodných hesel.

6.3.1 Podporované souborové systémy

Operační systémy typicky podporují práci s více souborovými systémy. Tuto podporu realizuje ovladač daného souborového systému. V OS Linux jsou ovladače pro souborové systémy dostupné v podobě modulů jádra. Pokud je požadavek na využití souborového systému, který jádro aktuálně neumí obsluhovat, je dynamicky nahrán příslušný modul. Pokud není tento modul k dispozici, tak nelze se souborovým systémem pracovat.

Informace o souborových systémech aktuálně podporovaných jádrem lze nalézt v souboru `/proc/filesystems`, viz výpis 6.1. První sloupec ve výpisu označuje, zda je souborový systém připojený k blokovému zařízení. Pokud ano, není zde uveden žádný text. Text *nodev* značí, že souborový systém není připojený k reálnému zařízení – jedná se o virtuální souborové systémy. Druhý sloupec označuje název takového souborového systému.

```
[]$ more /proc/filesystems
nodev   proc
nodev   sockfs
nodev   pipefs
nodev   usbfs
ext4
...
```

Výpis kódu 6.1: Podporované souborové systémy.

[9] Je zde také umístěn slovník využitý v počítačovém cvičení, které se zabývá prolomením hesla uživatelských účtů pomocí slovníkového útoku.

Ve výpisu jsou tedy vidět reálné a virtuální souborové systémy. Reálným systémem je `ext4`, který pracuje se žurnálem (viz konzistence dat a metadat). Virtuálními souborovými systémy jsou `proc`, `usbfs`, `pipefs` a `sockfs`.

Účelem souborového systému `usbfs` je zobrazovat připojená USB zařízení v podobě souborů, stejně jako je tomu u ostatních zařízení. Pomocí těchto souborů jsou pak USB zařízení dostupná v systému. Souborový systém `proc`, který je připojen do adresáře `/proc`, realizuje adresářovou strukturu souborů, ve kterých jsou dostupné informace o systému. Jedná se například o údaje o paměti a procesech. Tyto informace nejsou předem uloženy v souborech, které jsou fyzicky přítomné na úložném zařízení. Pokaždé, když se k těmto informacím přistupuje, souborový systém zajistí generování příslušných informací. Souborový systém `pipefs` je použit pro implementaci komunikace pomocí „rour“ (pipes) `|`. Například příkaz `ls -al | head` vytvoří nový soubor pro propojení výstupu prvního příkazu `ls` na vstup druhého příkazu `head`. Souborový systém `sockfs` udržuje informace o síťové komunikaci. Jeho účelem je přenášet data po síti pomocí zápisu/čtení ze souborů svázaných se sokety v tomto souborovém systému.

6.3.2 Úložná zařízení a diskové oddíly

Seznam dostupných úložných zařízení lze získat pomocí příkazu `fdisk`. Z výpisu 6.2 je patrno, že je v systému k dispozici jeden pevný disk realizovaný pomocí ploten a reprezentovaný souborem `/dev/sda`. Ve výpisu je také uveden počet cylindrů, hlav a velikost sektorů. U plotnových disků je každá plotna rozdělena na stopy a sektory. Plotny jsou uspořádány nad sebou. U velkých disků byla problematická vzdálenost mezi daty a metadaty. Metadata například udržují informace, kde jsou vlastní data fyzicky na disku uložena. Při přístupu k datům je tedy nejdříve nutno načíst metadata, zjistit polohu vlastních dat a pak přesunout hlavy pro přístup k datům. Vzdálenost mezi metadaty a daty se negativně projevuje časovými zdrženími při přesunu hlav mezi stopami na plotně. Tento problém se řeší rozdělením disků do cylindrických skupin. Každá cylindrická skupina obsahuje k vlastním datům i metadata, která jsou umístěna tak, aby nedocházelo k nadměrným přesunům hlav při přístupu k datům.

```
[]# fdisk -l
Disk /dev/sda: 1 000,2 GB, 1 000 204 886 016 bajtu
hlav: 255, sektoru na stopu: 63, cylindru: 121 601
Velikost sektoru (logickeho/fyzickeho): 512 bajtu / 512 bajtu
...

Zarizeni    Bloky       Id  System
/dev/sda1   107444224   83  Linux
/dev/sda2   53897216    82  Linux swap
/dev/sda3   815419392   83  Linux
```

Výpis kódu 6.2: Bloková zařízení v systému.

Pevný disk ve výpisu je rozdělen na tři oddíly. Oddíly jsou také svázány, stejně jako celé úložné zařízení, se soubory v adresáři `/dev`, které je reprezentují. Ve výpisu se jedná o soubory `sda1` až `sda3`. Dále lze ve výpisu spatřit počet bloků a typ oddílu (například Linux, Microsoft, NTFS/exFAT, atd). Druhý oddíl (swap) slouží jako odkládací

prostor, pokud není dostatek hlavní paměti. Tento odkládací prostor je využit pro realizaci virtuální paměti.

Ve výše uvedeném seznamu byly zobrazeny všechny přítomné oddíly. Ne všechny však musí být využívány v operačním systému – mohou být připojené či nepřipojené. Seznam připojených (používaných) oddílů do operačního systému a souborových systémů na nich použitých lze nalézt v souboru `/proc/mounts`, viz výpis 6.3. Každý řádek obsahuje informace o jednom připojeném oddílu.

```
[]$ more /proc/mounts
proc         /proc       proc        rw,... 0 0
/dev/sda1    /           ext4        rw,... 0 0
none         /selinux    selinuxfs   rw,... 0 0
none         /vbusbfs    usbfs       rw,... 0 0
```

Výpis kódu 6.3: Připojené souborové systémy na diskových oddílech.

Nejprve je uveden název souboru, který diskový oddíl reprezentuje – např. `/dev/sda1`; neplatí pro virtuální souborové systémy – např. `usbfs`. Další položkou je adresář, ve kterém je zpřístupněn připojený oddíl. Následující položka označuje souborový systém na připojeném oddílu. Dále jsou uvedeny parametry, které byly zadány při připojení souborového systému. Je zde uveden například typ přístupu (`rw` – čtení/zápis; `ro` — pouze čtení). Poslední informací jsou dvě čísla. Jejich hodnoty (0 – vypnuto, 1 – zapnuto) udávají, zda se bude připojený systém zálohovat a zda se připojený systém bude kontrolovat při následujícím startu operačního systému. Komplexní přehled o aktuálním stavu volného místa na připojených oddílech lze získat příkazem `df`, viz výpis 6.4.

```
[]$ df -h
Souborovy system  Velikost  Uzito  Volno  Uziti  Pripojeno do
/dev/sda1         150G      15G    36G    30%    /
/dev/sda3         407G      170G   238G   42%    /home
```

Výpis kódu 6.4: Stav využití připojených oddílů.

6.3.3 Metadata v souborových systémech

Metadata souborů v OS Linux jsou ukládaná ve struktuře nazvané i-uzel (i-node). I-uzel obsahuje metadata o souboru – typ souboru, vlastníka, skupinu vlastníka, přístupová práva, délku souboru, čas vytvoření, čas posledního přístupu a čas poslední modifikace. Dále i-uzel obsahuje ukazatele na bloky, ve kterých jsou vlastní data souboru. Malé soubory jsou adresovány pomocí přímých odkazů. Velké soubory jsou adresovány pomocí nepřímých odkazů.

Jednotlivé položky adresáře obsahují jméno souboru a číslo jeho i-uzlu – adresář tedy obsahuje informace potřebné pro nalezení dat svých souborů. V různých adresářích může být více stejných jmen, která ukazují na stejný soubor (i-uzel). Také může být ve stejném adresáři více různých jmen, která ukazují na stejný soubor. Tyto označujeme jako pevné odkazy (hard links). Pokud je smazán pevný odkaz na soubor, tak počet pevných odkazů na tento soubor je snížen o 1. Pokud je počet pevných odkazů na soubor roven 0, tak je soubor smazán. Dalším typem odkazu je symbolický odkaz. V tomto případě soubor

odkazuje na jiný soubor. Při smazání souboru, na který se symbolický odkaz odkazuje, nedochází k smazaní souboru se symbolickým odkazem. Podobně smazání symbolického odkazu ukazujícího na soubor tento soubor nesmaže.

Číslo i-uzlu určitého souboru lze zjistit na základě příkazu uvedeného ve výpisu 6.5. Jedná se o první sloupec ve výpisu. Ve výpisu je uveden i symbolický odkaz, který rozeznáme pomocí znaku `->`, který je mezi odkazem a souborem, na který odkaz odkazuje. U symbolických odkazů jsou nastavena plná práva, ovšem při práci se souborem, na který odkaz ukazuje, platí práva tohoto souboru.

```
[]$ ls -li /etc/passwd /etc/group /etc/rc
8740945  -rw-r--r--  1  root  root  /etc/group
8742639  -rw-r--r--  1  root  root  /etc/passwd
8741414  lrwxrwxrwx  1  root  root  /etc/rc -> rc.d/rc
```

Výpis kódu 6.5: Čísla i-uzlů.

Každý adresář obsahuje vždy dvě výchozí položky. První z nich je „.“ (tečka), která odpovídá i-uzlu samotného adresáře[10]. Druhá položka je „..“ (dvě tečky) a představuje odkaz na i-uzel „rodiče“ adresáře – na adresář nejblíže výše směrem ke kořenovému adresáři. Výjimkou je pouze kořenový adresář (root), pojmenovaný „/“. V tomto adresáři je odkaz „..“ odkazem opět na kořenový adresář. Příklad výchozích položek v adresáři je uveden na výpisu 6.6.

```
[]$ ls -ali
1703937  drwxr-xr-x.  komosny  komosny  .
2        drwxrwxrwx.  root     root     ..
9175041  drwxrwxr-x.  komosny  komosny  soubor
```

Výpis kódu 6.6: Automatické položky v adresáři.

6.3.4 Speciální soubory

Speciální soubory mohou reprezentovat zařízení v systému. Čtením dat ze souboru čteme data ze zařízení, zápisem do souboru toto zařízení ovládáme (ukládáme data). Jsou tedy použity stejné operace jako pro práci se soubory, což přináší abstrakci při používání zařízení. Rozlišujeme speciální soubory pro bloková zařízení, kde základní jednotka pro práci je blok dat. Dále rozlišujeme znaková zařízení, kde základní jednotka pro práci je jeden znak. Bloková zařízení využívají vyrovnávací paměť.

Rozlišení, zda se jedná o obyčejný soubor, soubor jako adresář, soubor jako symbolický odkaz, speciální soubor reprezentující blokové nebo znakové zařízení, lze zjistit pomocí prvního písmene ve výpisu informací o souborech:

- `-` obyčejný soubor,
- `d` adresář (je také soubor),
- `l` symbolický odkaz (je také soubor),
- `c` znakový speciální soubor,

[10] Je to tedy odkaz sám na sebe.

- `b` blokový speciální soubor.

Seznam speciálních souborů v systému lze získat výpisem obsahu adresáře `/dev`, kde je připojen virtuální souborový systém, ve kterém mají zařízení umístěný odpovídající soubor, viz výpis 6.7. Zobrazené speciální soubory reprezentují tato zařízení (nebo jejich části v případě oddílů na disku):

- `/dev/sda` – celý disk,
- `/dev/sda1` – první oddíl na disku,
- `/dev/sda2` – druhý oddíl na disku,
- `/dev/null` – softwarové zařízení slouží jako černá díra, data zapsaná do tohoto zařízení jsou ztracena; typicky slouží pro zahazování nepotřebných hlášení běžícího programu,
- `/dev/zero` – softwarové zařízení slouží jako generátor nulových bajtů; využitelné například pro přepisování předchozích dat v paměti,
- `/dev/random` – softwarové zařízení slouží jako generátor náhodných čísel.

```
[]$ ls -l
brw-rw---- 1 root disk  sda
brw-rw---- 1 root disk  sda1
brw-rw---- 1 root disk  sda2
crw-rw-rw- 1 root root  null
crw-rw-rw- 1 root root  zero
crw-rw-rw- 1 root root  random
```

Výpis kódu 6.7: Speciální soubory reprezentující zařízení ve virtuálním souborovém systému.

Na závěr si ukážeme reálný příklad využití speciálních souborů ve virtuálním souborovém systému. Bude pracovat se softwarovým zařízením pro generování náhodných znaků. Pro tento účel lze využít dvě softwarová zařízení `/dev/random` a `/dev/urandom`. Reálným příkladem bude generování hesel.

Ke generování náhodného hesla využijeme zařízení `/dev/urandom`. Rozdíl mezi `/dev/random` je v tom, že zařízení `/dev/urandom` je rychlejší z důvodu, že nečeká (neblokuje se) na získání náhodnosti z činnosti operačního systému. Proto náhodná čísla generovaná pomocí `/dev/urandom` nejsou tak „náhodná“ jako čísla generovaná pomocí `/dev/random`, jelikož do generování nevstupuje aktuální činnost operačního systému.

Pro generování hesla využijeme příkaz `cat`, který slouží ke čtení dat ze souboru. Jelikož čtení znaků příkazem `cat` by bylo nekonečné (protože náhodná čísla jsou stále generována), tak čtení je omezeno pomocí příkazu `head`. Načtená data jsou zobrazena v hexadecimální podobě pomocí příkazu `xxd`, viz výpis 6.8.

```
[]$ cat /dev/urandom | head -c 10 |  xxd
._b*k..DUr
```

Výpis kódu 6.8: Využití speciálního souboru reprezentujícího softwarové zařízení pro generování náhodných znaků.

Jako heslo by šlo použít i sekvenci znaků uvedenou ve výpisu. Tento výsledek však není uživatelsky přívětivý. Proto provedeme úpravu hesla.

1. Generované znaky budou převedeny na písmena „a“ až „z“ a to velká i malá. To zařídí příkaz `tr`, který provádí náhradu znaků podle zadané množiny. Parametrem příkazu jsou dvě zadané množiny, které specifikují zvolené rozsahy. Příkaz provádí náhradu znaků z první množiny na znaky z druhé množiny (dle jejich pořadí).
2. Definice počtu znaků hesla. K tomu bude využit příkaz `fold`, který provádí seskupování znaků.
3. Úprava počtu generovaných hesel. Zde bude využit příkaz `head`, který specifikuje počet řádků k zobrazení. Každý řádek odpovídá jednomu heslu.

Ve výpisu 6.9 lze vidět vygenerovaná náhodná hesla. Parametrem příkazu `tr` je množina možných znaků 'A-Za-z', které budou předány dále ze vstupu (jiné budou zahozeny). Pokud bychom chtěli například generovat PIN, tak by parametrem příkazu bylo '0-9' (jiné znaky mimo čísla budou zahozeny). Obdobně můžeme generoval znaky a čísla pomocí '0-9A-Z'. Parametrem příkazu `fold` je počet znaků k seskupení, tj. hesla mají po pěti znacích. Parametrem příkazu `head` je počet řádků k zobrazení, tj. počet hesel ke generování.

```
[]$ cat /dev/urandom | tr -cd 'A-Za-z' | fold -w 5 | head -n 5
PBsJm
SVvng
pVNbw
Hebsg
xdAMp
```

Výpis kódu 6.9: Využití speciálního souboru pro generování náhodných hesel.

Takto náhodně generovaná hesla lze například využít v situaci, kdybychom zaváděli nového uživatele do systému (nebo celou skupinu uživatelů). Uvedeným způsobem lze vygenerovat náhodné heslo, které je nastaveno jako výchozí pro vytvořený uživatelský účet (pomocí příkazu `passwd`). Uživateli vytvořeného účtu je toto heslo následně předáno. Uživatel pak může (by měl) toto náhodné heslo změnit za své vlastní.

7 SÍŤOVÝ SYSTÉM

Tato kapitola se zabývá síťovou částí operačních systémů. Jsou uvedeny části síťové komunikace nutné k pochopení principů operačních systémů a k popisu jejich služeb.

Pro síťovou komunikaci se využívají protokoly, které definují pravidla přenosu a formát dat. Typicky spolupracuje celá sada protokolů. Každý protokol z této sady plní určité funkce. Různé požadavky na komunikaci lze zajistit použitím různých protokolů. Protokoly pracují s bloky dat. Na vysílací straně se k datovým blokům připojují informace pro zajištění přenosu. Těmto informacím se říká **režijní data** a přidávají se do záhlaví (před přenášená data) nebo do zápatí (za přenášená data). Přijímací strana tato režijní data odebere a zpracuje.

Komunikace probíhá pomocí dvou základních způsobů:

- Spojově orientovaný přenos – před přenosem se sestavuje spojení a po přenosu se spojení ukončuje.
- Nespojově orientovaný přenos – neprovádí sestavování spojení; data jsou posílána bez předešlé komunikace s cílovou stanicí.

Spojově orientovaný přenos typicky poskytuje **spolehlivý** přenos, který garantuje správné doručení[1] zaslaných dat (pokud data nelze doručit, je o této skutečnosti informována aplikace odesílající data). Nespojově orientovaný přenos poskytuje **nespolehlivou** komunikaci ve významu „povolené“ ztráty dat při přenosu.

Z pohledu pozice v komunikačním řetězci rozlišujeme tyto typy zařízení:

- Koncová zařízení (end devices) – jsou zdrojem a příjemcem dat. Například se jedná o server, domácí PC (desktop) a mobilní zařízení.
- Mezilehlá zařízení (intermediate devices) – zajišťují přenos dat v síti. Jejich úlohou je nalezení cesty od zdroje k cíli (směrování), hledání záložních cest, detekce/oznamování chyb při přenosu a řešení priorit přenosů. Například se jedná směrovač, přepínač a firewall.

Všechna zařízení pro svou činnost využívají operační systémy, které mohou být obecné nebo přímo přizpůsobené danému typu zařízení (například Cisco IOS).

Výše uvedený popis je pouze stručným shrnutím základních pojmů síťové komunikace pro snadný vstup do dalších kapitol. Dále je již pozornost věnována vybraným částem důležitým pro operační systémy[2].

7.1 Implementace sítě

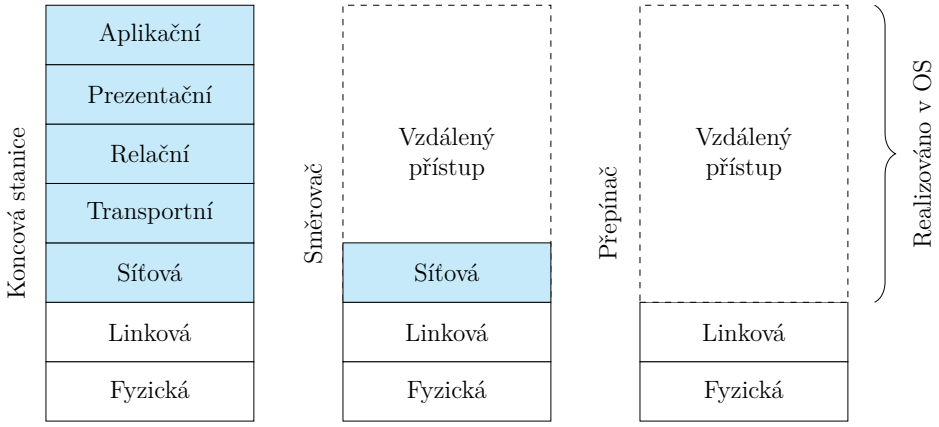
Z pohledu komunikace operační systém realizuje vrstvy od síťové až po aplikační dle referenčního modelu OSI (model není vázán na konkrétní protokoly). Tyto vrstvy jsou

[1] Pod pojmem „správné“ doručení je zde myšleno zajištění doručení všech zaslaných dat a také jejich doručení ve stejném pořadí, v jakém byla vyslána.

[2] Což je účelem tohoto předmětu. Komplexní informace o síťové komunikaci jsou probírány v jiných předmětech studijních programů.

zachyceny na obrázku 7.1 (barevně označeny u koncové stanice). Operační systém tedy zajišťuje (podle definice jednotlivých vrstev):

- Rozšíření o síťovou část – aplikační vrstva.
- Koordinaci formátu přenášených dat – prezentační vrstva.
- Řízení a synchronizaci průběhu komunikace – relační vrstva.
- Přenos dat mezi procesy – transportní vrstva; datové jednotky na této vrstvě jsou nazývány segmenty/datagramy.[3]
- Přenos dat mezi zařízeními – síťová vrstva; datové jednotky na této vrstvě jsou nazývány pakety.



Obrázek 7.1: Vrstvy komunikace realizované operačním systémem.

Nejnižší dvě vrstvy (linková a fyzická) jsou realizovány v síťových rozhraních NIC (Network Interface Controller) pomocí hardware a firmware. Síťových rozhraní NIC může být v rámci operačního systému několik. Operační systém koncových zařízení (server/klient) zahrnuje všech pět vrstev. U síťových prvků operační systém zahrnuje vrstvy podle toho, jaké funkce poskytuje. Na obrázku 7.1 jsou zobrazeny vybrané typy zařízení a vrstvy, které implementuje operační systém na těchto zařízeních. Pokud síťový prvek poskytuje rozšířené funkce mimo svou základní činnost (přepínání, směrování), tak disponuje operačním systémem implementujícím všechny vrstvy. Tato rozšířená funkce může být například vzdálený přístup, kdy síťový prvek vystupuje jako server.

Operační systém implementuje tyto protokoly podle modelu TCP/IP (pouze vybrané):

- Na transportní vrstvě TCP a UDP – mohou být poskytovány jako služby jádra.
- Na aplikační vrstvě FTP, HTTP, TELNET, TFTP, SSH – jsou poskytovány v podobě serverových procesů.

Pro procesy operačního systému je stěžejní transportní vrstva. Tato provádí propojení komunikace s cílovými a zdrojovými procesy (programy) pomocí **portů** [34]. Jedná se o 16bitové číslo, které označuje síťovou službu v operačním systému. Čísla portů přiděluje

[3]Segment pokud hovoříme o bloku dat transportní vrstvy obecně, nebo pokud hovoříme o protokolu TCP. U protokolu UDP označujeme datové jednotky jako datagramy.

organizace IANA (Internet Assigned Numbers Authority). Rozlišujeme tři rozsahy čísel portů [37]:

- **Systémové porty** (0–1023). Jsou rezervovány známé síťové služby (např. port 69 pro TFTP). Označují se také jako známé nebo privilegované porty, protože tyto porty mohou používat pouze procesy (realizující tyto služby) spuštěné v privilegovaném režimu. Používají se na straně serveru.
- **Uživatelské porty** (1024–49151). Jedná se o služby, které si registrují jejich tvůrci. Označují se proto také jako registrované. Jsou přidělovány typicky na straně serveru.
- **Dynamické/privátní porty** (49152–65535). Jsou přidělovány službám dynamicky na straně klienta; nejsou vázány na konkrétní službu.

Hodnoty známých a registrovaných portů lze nalézt v operačních systémech UNIX a Linux v souboru `/etc/services`. Tento soubor navazuje čísla portů na název příslušné služby. Každému portu odpovídá jeden řádek ve tvaru název služby a port/protokol na transportní vrstvě. Služby se stejným portem rozlišuje protokol.

Výpis 7.1 zobrazuje příklady systémových portů a svázaných služeb, které jsou probírány v rámci těchto skript. Druhý výpis 7.2 zobrazuje příklad uživatelských portů. Jedná se o službu VNC pro přístup ke vzdálené ploše. Pozorní studenti si možná všimnou hry s registrovaným portem, ta však dále probírána není.

```
[]$ more /etc/services
...
ftp-data      20/tcp
ftp-data      20/udp
ftp           21/tcp
ftp           21/udp
ssh           22/tcp
ssh           22/udp
telnet        23/tcp
telnet        23/udp
...
```

Výpis kódu 7.1: Příklady systémových portů.

```
[]$ more /etc/services
...
blizwow       3724/tcp #World of Warcraft
blizwow       3724/udp #World of Warcraft
...
rfb           5900/tcp #Remote Framebuffer(VNC)
rfb           5900/udp #Remote Framebuffer(VNC)
```

Výpis kódu 7.2: Příklady uživatelských portů.

Práce s porty je následující: Klient komunikuje pomocí dynamického portu – třetí kategorie. Tento port je použit pouze po dobu spojení. Při navázaní spojení na službu serveru, zašle data na systémový nebo uživatelský port – první a druhá kategorie. Hodnota dynamického portu (na straně klienta) je serveru známa ze záhlaví TCP segmentů či UDP datagramů.

Operační systém může disponovat více síťovými rozhraními NIC. Každé z těchto rozhraní má adresu na síťové vrstvě (vrstvě internetu TCP/IP) – IP adresu[4]. Pro protokol IP ve verzi 4 se jedná o 32bitové číslo, verze 6 používá 128bitové číslo. Přidělovaní bloků IP adres (organizacím/ISP) je řízeno organizací IANA (Internet Assigned Numbers Authority). Tato organizace alokuje souvislé bloky IP adres pěti regionálním pobočkám. Evropa spadá pod pobočku RIPE NCC (RIPE Network Coordination Centre). Tyto regionální pobočky dále přidělují rozsahy IP adres poskytovatelům služeb – organizacím a ISP (Internet service provider). Delegování adresního prostoru může být buď přímé, nebo prostřednictvím dalších složek – LIR (Local Internet Registry) nebo NIR (National Internet Registry). Organizace nebo ISP pak přiděluje konkrétní IP adresu danému zařízení.

Alokované rozsahy IP adres organizacím/ISP jsou vedeny v registrační databázi. Informace z této databáze lze získat například pomocí protokolu WHOIS, který v OS Linux využívá stejnojmenný příkaz `whois`. Příklad získání rozsahu IP adres přidělených VUT zobrazen na výpisu 7.3.

```
[]$ whois 147.229.147.1
...
inetnum:      147.229.0.0 - 147.229.254.255
netname:      VUTBRNET
descr:        Brno University of Technology
country:      CZ
address:      Brno University of Technology
address:      Antoninska 1
address:      601 90 Brno
address:      The Czech Republic
```

Výpis kódu 7.3: Příklad přístupu do registrační databáze rozsahů IP adres.

7.2 Sokety a servery

Soket[5] je vstupně/výstupní bod operačního systému do síťové komunikace. Jde o rozšíření rour (pipe) pro komunikaci mezi procesy.

Soket zahrnuje adresu síťové i transportní vrstvy. Pomocí IP adresy specifikuje koncovou stanici. Port společně s protokolem určuje síťovou službu – proces. Pozice soketu v rámci operačního systému je uvedena na obrázku 7.2 [11]. Na obrázku jsou naznačeny dva procesy komunikující na úrovni služby.

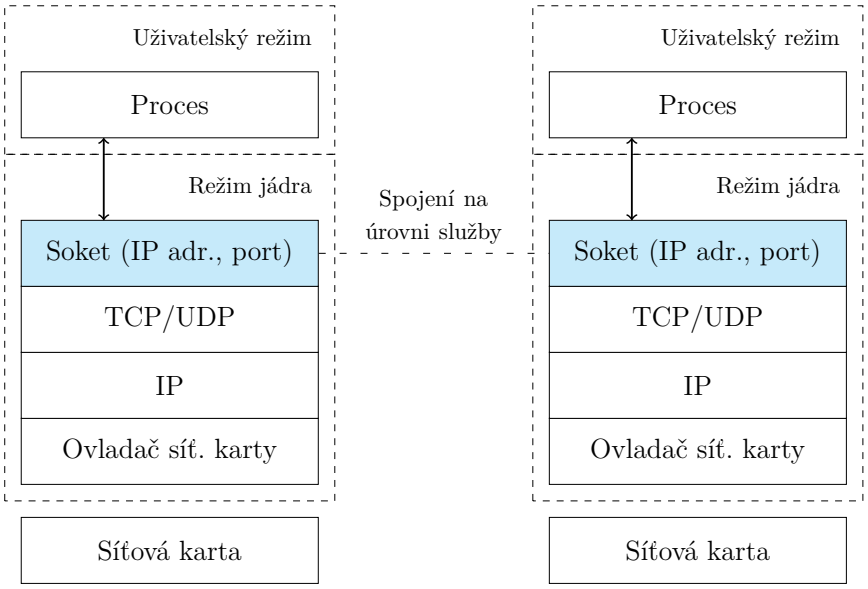
7.2.1 Stavy komunikace

Sokety při spojově-orientované komunikaci procházejí různými stavy, kterými jsou:

1. čekání na spojení,
2. sestavování spojení,

[4]Zařízení může mít několik IP adres. Například směrovač má přiděleno tolik adres, kolik má rozhraní.

[5]Anglické slovo „socket“ by se dalo přeložit do češtiny jako „schránka“ Tento pojem však není rozšířený. V rámci skript bude proto použit rozšířený název soket.

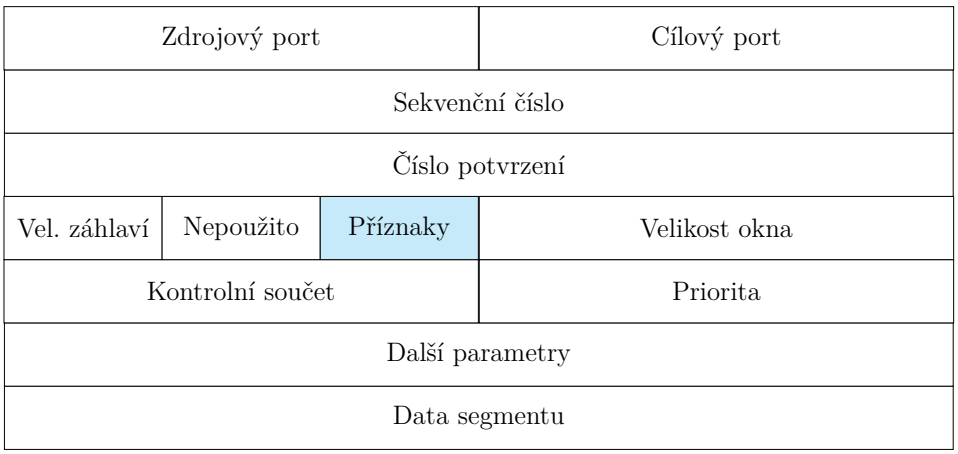


Obrázek 7.2: Pozice soketu v rámci operačního systému.

3. přenos dat,
4. ukončování spojení.

Operační systém na stav soketů reaguje spouštěním nových procesů pro obsluhu příchozích požadavků, aplikací bezpečnostní politiky atd.

Přechody mezi stavy jsou řízeny pomocí **příznaků** v záhlaví zasílaných segmentů protokolu TCP. Struktura záhlaví segmentu je zobrazena na obrázku 7.3.



Obrázek 7.3: Příznaky v segmentu TCP pro řízení stavu soketů v operačním systému.

Základními příznaky pro řízení stavu soketů jsou:

- ACK – potvrzení přijatých dat.
- PSH – přenos dat.

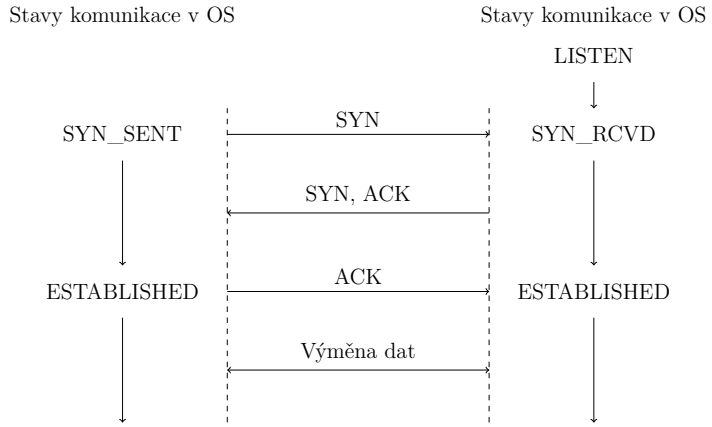
- SYN – žádost o sestavení spojení.
- FIN – ukončení spojení.

Pro sestavení spojení jsou použity příznaky SYN a ACK. Pro ukončení spojení jsou použity příznaky FIN a ACK. Pro přenos dat se používají příznaky PSH a ACK.

Spojení protokolu TCP jsou jednosměrná, proto se pro obousměrnou komunikaci používají dvě jednosměrná spojení. Spojení se klasicky sestavuje pomocí tří segmentů[6].

Sestavení spojení

První jednosměrné spojení se zahájí pomocí zaslání segmentu s nastaveným příznakem SYN. Naslouchající soket na straně serveru tento segment přijme a potvrzuje přijatou žádost na soket klienta – zaslaný segment má nastaven příznak ACK. Dále tento segment zasílá žádost o sestavení druhého jednosměrného spojení pomocí nastaveného příznaku SYN. Třetí segment potvrzuje navázání druhého jednosměrného spojení, viz obrázek 7.4.



Obrázek 7.4: Stavy komunikace v operačním systému – sestavení spojení.

Při sestavování spojení může být soket na straně serveru ve stavu LISTEN (poslouchá/čeká na příchozí spojení) nebo SYN_RCVD (přijal od klienta první segment SYN). Soket na straně klienta se může nacházet ve stavu SYN_SENT (poslán segment SYN). Po navázání spojení přejdou oba sokety do stavu ESTABLISHED.

Přenos dat

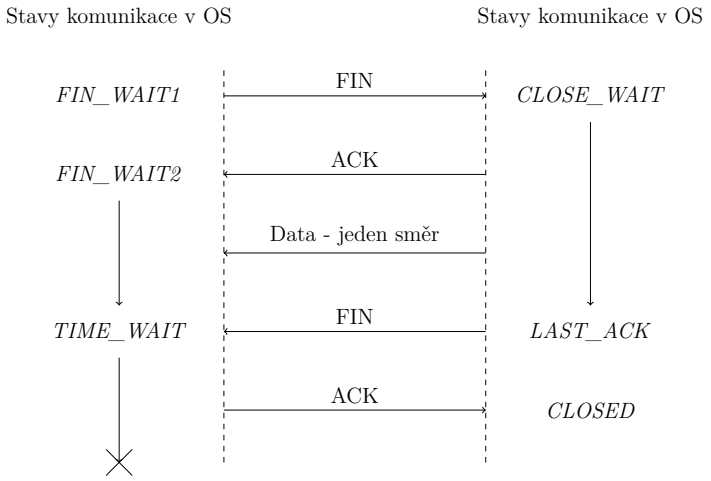
TCP segmenty nesoucí data mají nastaven příznak PSH. Segment s příznakem ACK slouží pro potvrzení přijetí dat.

Ukončení spojení

Pro ukončení spojení slouží segment s příznakem FIN. Ukončení může provést server i klient, a to nezávisle. Pokud zašle segment s příznakem FIN pouze jedna strana, tak dojde k ukončení pouze jednoho jednosměrného spojení. Druhá strana může ještě posílat data,

[6] Anglicky je tato procedura nazývána „three-way handshake“ – třikrát potřesení rukou.

dokud sama neukončí druhé jednosměrné spojení dalším segmentem s příznakem FIN. Přijetí segmentů s příznakem FIN je protistranou potvrzováno. Při ukončování spojení prochází sokety několika stavy, které jsou zobrazeny na obrázku 7.5.



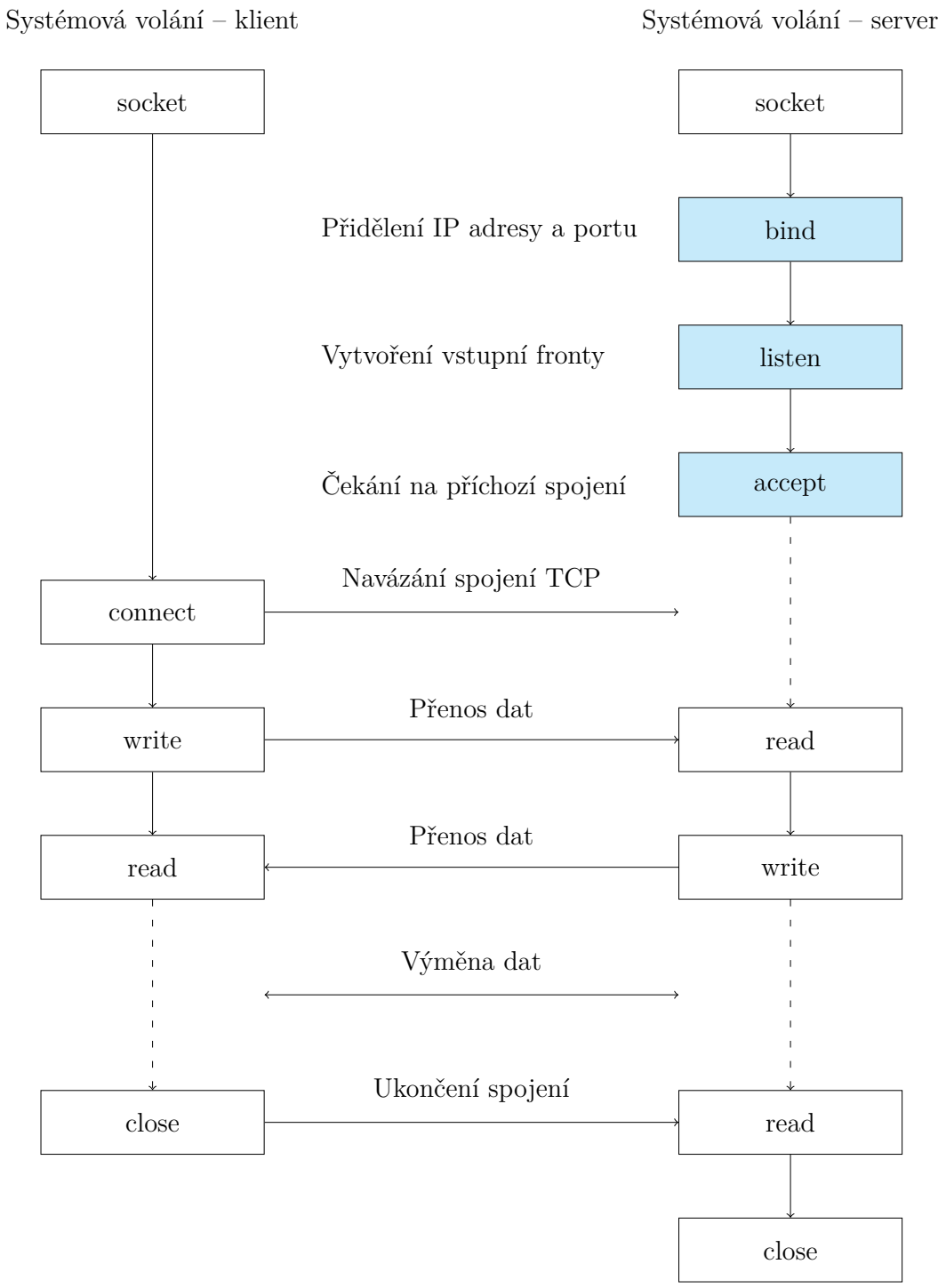
Obrázek 7.5: Stavy komunikace v operačním systému – ukončení spojení.

Tyto stavy jsou:

- FIN_WAIT1 – ukončení prvního spojení.
- CLOSE_WAIT – příjem segmentu s příznakem FIN (pro první spojení).
- FIN_WAIT2 – příjem potvrzení o ukončení prvního spojení, tento stav trvá do doby, než protistrana ukončí druhé spojení.
- LAST_ACK – ukončení druhého jednosměrného spojení.
- TIME_WAIT – čekání na potvrzení ukončení druhého spojení. Tento stav typicky trvá po dobu 2 minut (hodnota dle OS). Důvodem tohoto intervalu je skutečnost, že zaslané potvrzení se může ztratit a protistrana si pak vyžádá nové potvrzení. Toto by nebylo možné, kdyby byl soket již uzavřen.

7.2.2 Činnost soketů

Sokety se obsluhují pomocí systémových volání, kterými jsou: *socket*, *connect*, *bind*, *listen*, *accept*, *close*, *write* a *read*. Použití těchto systémových volání je zobrazeno na obrázku 7.6.



Obrázek 7.6: Systémová volání pro práci se sokety.

socket

Založí nový soket.

connect

Používá se na straně klienta pro navázání spojení. Navazování probíhá ve třech krocích pomocí příznaků SYN a ACK.

bind

Je použito na straně serveru a přiřadí soketu IP adresu a port. Pokud by nebylo použito, byl by soketu přidělen předem neznámý port z třetí kategorie. U serveru se očekává, že bude naslouchat na konkrétním portu svázaným s poskytovanou službou.

listen

Převede soket na straně serveru do pasivního režimu, tedy bude přijímat spojení (pomocí dalšího volání *accept*). Výstupem je vytvoření vstupní fronty pro příchozí spojení.

accept

Je voláno na straně serveru a slouží k vyzvednutí příchozího požadavku o spojení z vytvořené vstupní fronty (pomocí *listen*). Volání vytvoří nový soket. Pokud je zavoláno v době, kdy ve vstupní frontě není žádná žádost o příchozí spojení, tak je blokováno (než nějaká žádost přijde).

close

Uzavírá soket. Ukončení spojení se provádí pomocí příznaků FIN a ACK.

write* a *read

Pro přenos dat, použity segmenty s příznaky PSH a ACK.

Příklad stavu soketů je zobrazen ve výpisech 7.4 a 7.5. První výpis zobrazuje komunikaci procesů v síti (sokety domény Internet). Druhý výpis zobrazuje komunikaci procesů v rámci operačního systému (aktivní sokety domény UNIX).

```
[]$ netstat
Aktivni sokety domeny Internet
Proto  Recv-Q  Send-Q  Local_Address       Foreign_Address    State
tcp    0       0       127.0.0.1:631       0.0.0.0:*          NASLOUCHA
tcp    0       0       127.0.0.1:25        0.0.0.0:*          NASLOUCHA
tcp    0       0       89.71.174.89:36648  147.229.144.38:22  SPOJENO
tcp    1       0       127.0.0.1:35846     127.0.0.1:631      CLOSE_WAIT
tcp    0       0       :::22               :::*               NASLOUCHA
udp    0       0       0.0.0.0:631         0.0.0.0:*
```

Výpis kódu 7.4: Stavy soketů domény Internet.

```
[]$ netstat
Aktivni sokety domeny UNIX
Proto Citac Typ    Stav       I-Uzel Cesta
unix  2     DGRAM              1229   @/org/kernel/udev/udevd
unix  2     STREAM NASLOUCHA   7474   /tmp/.sockets/audio0
unix  20    DGRAM              6291   /dev/log
unix  3     STREAM SPOJENO     583062
```

Výpis kódu 7.5: Stavy soketů domény Unix.

Nejprve ke komunikaci v síti. Sloupec s označením *Proto* označuje použitý protokol, položky *Recv-Q* a *Send-Q* zobrazují počet bajtů ve vstupní, respektive výstupní frontě. *Local Address* je IP adresa síťového rozhraní (adresa na síťové vrstvě), za dvojtečkou je uvedeno číslo portu (adresa na transportní vrstvě). *Foreign Address* je IP adresa a port protilehlé stanice[7]. Poslední sloupec s názvem *State* označuje aktuální stav spojení. Lokální adresa 0.0.0.0 (:: pro IPv6) značí čekání na spojení na daném portu. Adresa 127.0.0.1 značí, že budou přijata jen spojení ze stejného počítače (localhost).

Dále jsou zobrazeny aktivní sokety domény UNIX. Soket použitý pro komunikaci procesů v rámci jednoho stroje nepoužívá IP adresu a port. Zde je jako adresa použito číslo i-uzlu (číslo struktury metadat popisující daný soubor, viz souborové systémy) souboru, který realizuje soket. Sloupec *Citac* zobrazuje počet procesů komunikujících přes soket.

7.2.3 Serverové procesy

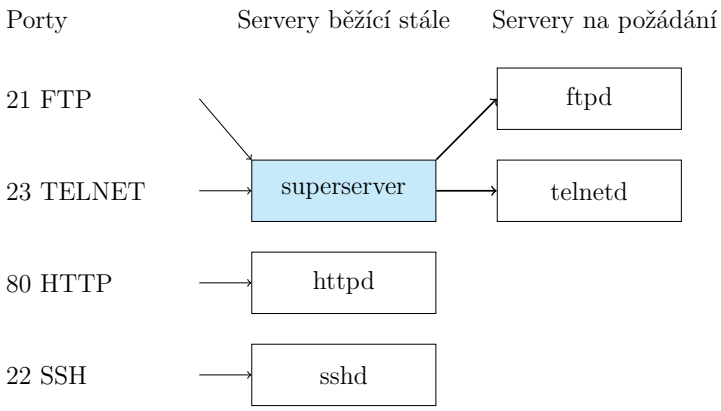
Služby operačního systému jsou zprostředkovány programy, které nazýváme servery. Slovo server používáme obecně pro „entitu“, která poskytuje síťovou službu (například server SSH či TELNET). Proces každého serveru má přiřazenu adresu na transportní vrstvě – port. Tato adresa rozlišuje jednotlivé servery v rámci operačního systému. Servery pracující na pozadí se označují jako **démoni** (daemon). Servery mohou být spouštěny dvojím způsobem:

- **Stále** – server je spuštěn při startu operačního systému; předpokládá se velké množství příchozích požadavků.
- **Na požádání** – server je aktivován po přijetí požadavku na využití dané služby; předpokládá se menší využití.

Stále spuštěné servery zabírají systémové prostředky. Spouštění serverů na požádání zajišťuje server, který běží stále. Tento server je označován jako **superserver** (superdémon).

Příklad použití superserveru je zobrazen na obrázku 7.7. Superserver běží stále a naslouchá na portech serverů FTP (`ftpd`) a TELNET (`telnetd`), které jsou spouštěny na požádání. V případě příchozího požadavku na některý z těchto portů aktivuje superserver příslušný server. Oproti tomu servery HTTP (`httpd`) a SSH (`sshd`) jsou navázány přímo na příslušné porty a běží stále.

[7]Port 631 náleží protokolu IPP – Internet Printing Protocol. Protokol IPP používá protokol TCP i UDP. Port 22 náleží protokolu SSH (Secure Shell).



Obrázek 7.7: Příklad serverů běžících stále a spouštěných na požádání.

Uveďme si použití superdémona pro spuštění málo využívané služby TELNET. V adresáři `/etc/xinetd.d/` jsou uvedeny konfigurační soubory pro jednotlivé služby. Konfigurační soubor pro službu TELNET `/etc/xinetd.d/telnet` je uveden ve výpisu 7.6. Položky v konfiguraci mají tento význam: *user* – uživatel, pod kterým se služba spustí; *server* – cesta k programu serveru a *disable* – zda je spouštění služby zakázáno.

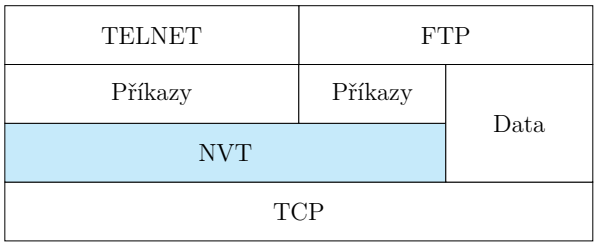
```
service telnet
{
user            = root
server          = /usr/sbin/in.telnetd
disable         = yes
}
```

Výpis kódu 7.6: Konfigurace služby TELNET spouštěné pomocí superdémona.

7.3 Vzdálené připojení

Základními službami pro vzdálené připojení jsou TELNET a SSH. Po přihlášení k serveru se místní počítač začne chovat jako terminál připojený ke vzdálenému počítači.

TELNET je spolehlivá služba. Pracuje nad protokoly TCP a NVT (Network Virtual Terminal), který zajišťuje prezentaci dat, tj. převod přenášených znaků na bajty a obráceně, viz 7.8.



Obrázek 7.8: Protokol NVT a jeho využití pro služby TELNET a FTP.

Služba TELNET je nezabezpečená, je tedy nevhodná pro použití ve veřejných sítích. Naproti tomu služba SSH poskytuje zabezpečenou komunikaci pomocí **šifrování** přenášených dat a provádí také **autentizaci** komunikujících stanic a uživatelů [8]. Mimo přenosu příkazů umožňuje i přenos dat – kombinuje tedy funkci nezabezpečených služeb FTP a TELNET. Další výhodou je možnost vytváření tzv. **tunelů** mezi dvěma počítači. Tyto tunely se používají při komunikaci založené na nezabezpečených protokolech. Nezabezpečená komunikace je přesměrována do vytvořeného tunelu a dochází tak k jejímu dodatečnému zabezpečení. Zabezpečení komunikace samozřejmě přináší vyšší požadavky na přenosovou kapacitu sítě.

Základní pojmy, které se vztahují ke službě SSH:

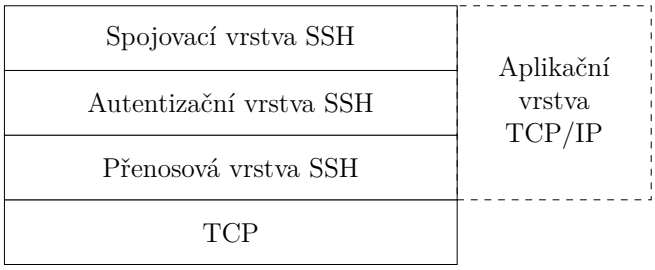
- Autentizace – ověření **identity** (ověření, zda je uživatel/systém tím, za koho se vydává). Každé spojení přes SSH vyžaduje dvě autentizace. Klient ověřuje identitu serveru a naopak server ověřuje identitu uživatele.
- Integrita – neměnnost dat při přenosu přes síť. Pokud třetí entita vstoupí do relace a pozmění přenášená data, je porušena integrita dat.
- Šifrování – činí přenášená data nečitelná pro kohokoliv vyjma příjemce.

Služba SSH umožňuje autentizaci pomocí **veřejných a privátních klíčů**. Uživatel má často více účtů na více serverech (pro více služeb). Při přístupu ke každému z těchto účtů toto heslo zadává. Doporučuje se, aby každý účet měl jiné heslo – z důvodu možné kompromitace dalších účtů při prolomení jednoho hesla. To ovšem přináší těžkopádnou správu hesel. Čím častěji jsme nuceni zadávat hesla, tím se zvyšuje pravděpodobnost chyby a prozrazení hesla[8]. S použitím veřejných a privátních klíčů se redukuje počet zadávání hesel. Výhodou je také připojení bez manuálního zadání hesla, což se využívá pro automatizaci přihlašování na vzdálené stanice (viz cvičení – přenos dat).

Princip veřejného a tajného klíče je založen na faktu, že data šifrujeme veřejným klíčem, ale s pomocí tohoto klíče je nelze dešifrovat – jedná se o jednosměrnou operaci. Data lze dešifrovat pouze privátním klíčem. Pokud se chceme přihlašovat pomocí klíčů, tak na server umístíme svůj veřejný klíč. Program, který provádí správu tajných klíčů na klientské stanici se nazývá **autentizační agent** [5]. Tyto klíče se používají pro přihlášení na různé servery.

Organizace služby SSH je zobrazena na obrázku 7.9 [14]. Přenosová vrstva provádí autentizaci serveru. Dále provádí šifrování, kontrolu integrity a volitelně kompresi dat. Autentizační vrstva provádí ověření klienta pomocí hesla nebo veřejného klíče. Spojovací vrstva definuje přenosové kanály. Jedno spojení SSH může zahrnovat několik kanálů, pomocí kterých jsou poskytovány koncové služby SSH. Například se jedná pro přesměrování portů pro vytvoření tunelu.

[8]Například zápis hesla místo uživatelského jména. Na některých systémech se zaznamenává text zadaný do terminálu, heslo se tak může objevit v logu.



Obrázek 7.9: Vrstvy služby SSH.

Všechny části SSH jsou zahrnuty v aplikační vrstvě protokolového modelu TCP/IP. Z pohledu referenčního modelu OSI, který dále rozlišuje relační, prezentační a aplikační vrstvu, můžeme dílčí části SSH zařadit následovně:

- Aplikační – příkazy terminálu a přenos souborů.
- Prezentační – šifrování.
- Relační – vytvoření/ukončení spojení a přesměrování portů.

Pro SSH existuje několik implementací. Jednou z nejrozšířenějších je OpenSSH, která zahrnuje několik programů s tímto účelem:

- SSH klient (*ssh*) – přihlašování na straně klienta.
- Secure Copy (*scp*) – kopírovaní dat.
- Secure File Transfer (*sftp*) – sdílení a stahování souborů.
- SSH server (*sshd*) – server (démon).

7.4 Bezpečnost síťového systému

Pokud dojde k narušení ochrany operačního systému, mohou nastat důsledky v podobě ztráty soukromí či zcizení informací, což může v komerční sféře odpovídat značným finančním ztrátám. V této kapitole jsou probrány základy ochrany operačních systémů.

7.4.1 Základy bezpečnosti

Bezpečnost operačních systémů závisí na **zranitelnosti** systému včetně jeho uživatelů, což je možnost přimět operační systém (uživatele) k nestandardnímu chování. Zranitelnost systému závisí na zranitelnosti prvků, kterými je systém tvořen. Z technického hlediska se může se jednat o samotné jádro, systémové a aplikační programy; viz struktura operačního systému na obrázku 3.1. Hrozbou pro operační systémy jsou lidé, kteří mají znalosti o jeho funkci nebo dokážou vhodně manipulovat s uživateli.

Vhodným přístupem, jak zajistit ochranu systému je „vžít se do role útočníka“ a hledat slabá místa jeho pohledem (tzv. White Hat). Obecný postup útočníka je následující:

1. Sběr informací – Útok je typicky zahájen pasivním a/nebo aktivním získáváním informací. Mezi tyto informace patří například uživatelská jména a hesla, typ a

verze operačního systému, verze provozovaných služeb a znalost adresace (porty, IP adresy). Příklady způsobů pro sběr informací jsou sociální inženýrství (manipulace uživatelů), testování (skenování) portů, odposlouchávání a analýza síťového provozu. Při znalosti technických informací lze dohledat chyby ve verzích operačních systémů a služeb ve veřejných databázích. Na základě těchto chyb pak zvolit typ útoku.

2. Získání přístupu – Jestliže útočník získá neprivilegovaný přístup, může dále získat přístup privilegovaný. To platí zvlášť v případě, kdy je bezpečnostní politika operačního systému benevolentní a umožňuje běžnému uživateli provádět aktivity, které pro jeho práci nejsou nezbytně nutné.
3. Získání utajovaných informací a zneužití systému – Pomocí privilegovaného nebo i neprivilegovaného přístupu může útočník získat citlivé informace. Dále může provést instalaci tzv. zadních vrátek (back-door), které umožní útočníkovi opětovný přístup do systému. Často se jedná otevřené porty služeb operačního systému. Operační systém tak může být zneužit pro různé účely.

Mezi základní útoky spadá prolomení uživatelských účtů, které může být realizováno pomocí sociálního inženýrství nebo pomocí programového útoku.

V případě **sociálního inženýrství** [18, 25] útočník využívá lidského faktoru. Někdy je jednodušší oklamat uživatele operačního systému, než provést technický útok. Často se využívá podvržených informací (ať už v ústním podání, v podobě falešných průkazů a dokumentů, či jakkoli jinak) a znalosti citlivých informací o dotčené osobě, které zvyšují důvěryhodnost předstírané identity. Útočník je často schopen získat informace přesvědčením oběti, že tato data potřebuje a má právo je znát. V podstatě je tento útok založený na důvěřivosti oběti a momentu překvapení, při kterém oběť přestane racionálně uvažovat.

Další útok je testování neznámého hesla oproti slovům ve slovníku, tzv. **slovníková metoda**. Slovníkové útoky mohou být úspěšné, protože uživatelé používají jednoduchá hesla, která obsahují známá slova. Naproti tomu útok **hrubou silou** zkouší různé kombinace znaků a ty testují, zda jsou použity jako heslo. Tento útok postupně zkouší všechny možné kombinace písmen, čísel a znaků, dokud není nalezeno heslo. Nevýhodou této metody je velká časová náročnost. Příklad tohoto útoku je zachycen ve výpisech 7.7 a 7.8. V prvním případě jsou zachyceny neúspěšné pokusy o přihlášení pro uživatele *root*. Ve výpisu lze vidět i adresu odkud útok probíhá. Pomocí poziční databáze IP adres lze následně dohledat přibližnou polohu zdroje útoku. Ve druhém případě probíhá útok na uživatele *sameer*, který však v systému neexistuje. Někdy je pro útočníka výhodnější zkoušet typická jména uživatelů a pro tato jména hledat hesla. Tato taktika vychází z předpokladu, že uživatel *root* je znalý a používá dobré (silné) heslo. Běžní uživatelé v systému takto znalí nemusí být a jejich heslo může být snazší k prolomení.

```
sshd: Failed password for root from 183.3.202.190 port 23515 ssh2
sshd: Failed password for root from 183.3.202.190 port 23515 ssh2
sshd: Failed password for root from 183.3.202.190 port 23515 ssh2
sshd: Received disconnect from 183.3.202.190: 11:  [preauth]
sshd: PAM 2 more authentication failures
```

Výpis kódu 7.7: Útok na školní server – uživatel root.

```
sshd: Invalid user sameer from 121.241.34.137
sshd: Failed password for invalid user sameer from 121.241.34.137
sshd: Received disconnect from 121.241.34.137: 11: Bye Bye [preauth]
```

Výpis kódu 7.8: Útok na školní server – uživatel saamer.

Mezi další útoky patří použití škodlivého programu, který je zanesen do operačního systému. Těmito jsou:

- Virus – Kód přidaný k jinému programu. Je závislý na aktivitě uživatele (uživatel spustí program s virem).
- Červ – Samostatný program. Šíří se sám (nepotřebuje aktivitu uživatele).
- Trojský kůň – Program, který vypadá nebo se vydává za neškodnou aplikaci. Tyto programy bývají vystaveny na Internetu ke stažení zdarma.

Po instalaci operačního systému jsou jeho nastavení ve výchozím stavu. Tato nastavení jsou z pohledu ochrany nedostačující. Je potřeba provést další kroky, které jsou:

- Změnit výchozí uživatelská jména a hesla.
- Omezit přístup k systémovým zdrojům pouze pro uživatele, kteří jsou k příslušné činnosti autorizovány.
- Vypnout nepotřebné služby.

Bezpečnost je stav (nikoliv vlastnost), a tudíž na systém musí být stále dohlíženo pomocí tzv. **zabezpečovacího cyklu** [20, 25]: 1) zabezpečení, 2) monitorování, 3) testování a 4) zlepšení. Zabezpečení operačního systému se provádí opravou zranitelných částí systému – aplikováním opravných programů a nastavením ochrany před známými hrozbami. Monitorování zahrnuje detekci a potírání útoků. V testovací fázi se provádí útoky z pohledu útočníka. Testují se fáze zabezpečení a monitorování. Zlepšení zahrnuje analýzu údajů získaných z monitorovací a testovací fáze. Nalezená zlepšení jsou aplikována v kroku jedna – zabezpečení.

7.4.2 Firewall

Účelem firewallu je chránit operační systém před hrozbami ze sítě pomocí **filtrování komunikace**. Firewall pracuje na základě shody s pravidly, která jsou prohledávána sekvenčně. Typicky jsou tato pravidla aplikována v pořadí od konkrétních po obecná. Na konci je obvykle výchozí pravidlo, které všechen provoz nedotčený předchozími pravidly povolí nebo zakáže. Firewall může provádět i další činnosti, například změnu IP adresy nebo portů pro procházející komunikaci.

Firewall může být realizován jako software v rámci operačního systému nebo jako samostatný hardware. Další text se věnuje firewallu jako součásti operačního systému.

Firewally dělíme dle úrovně filtrování na paketové, stavové a aplikační [15, 25].

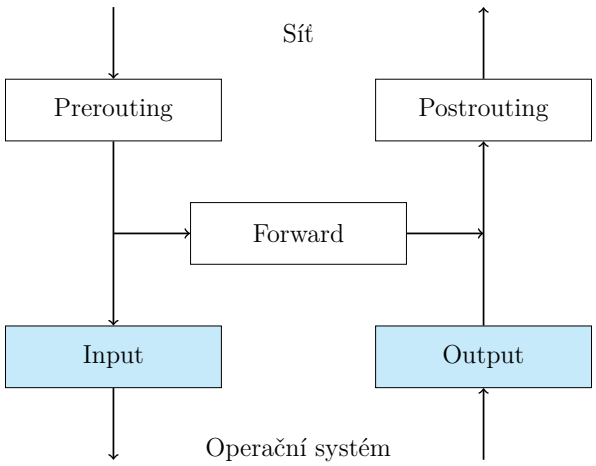
- Paketový – Nejjednodušší typ firewallu. Pro svou činnost využívá informací ze záhlaví datových bloků síťové a transportní vrstvy. Výhodou je vysoká rychlost zpracování. Poskytuje pouze základní úroveň ochrany.

- Stavový – Využívá navíc informace o stavu spojení, dokáže tak rozlišit zprávy nového (sestavovaného) spojení od zpráv již ustaveného provozu. Pracuje s příznaky v TCP segmentech (viz průběh sestavování spojení protokolu TCP). Může například stanovit, kterým směrem mohou být spojení zahájena. Poskytuje větší míru ochrany než paketový firewall.
- Aplikační – Označován také jako aplikační brána, proxy firewall a proxy server. Umožňuje kontrolu obsahu přenášených zpráv (např. filtrování nevhodného obsahu). Nevýhodou jsou vyšší hardwarové nároky. Poskytuje nejvyšší míru ochrany.

U aplikačního firewallu dále rozeznáváme dva typy:

- Netransparentní proxy – Komunikace probíhá tak, že klient pošle firewallu požadavek na otevření spojení se zvolenou službou a firewall toto spojení otevře. Uživatel se nepřipojuje přímo na server. Odpověď od serveru je zaslána firewallu a ten ji zprostředkuje klientovi. Všechna data tak jdou přes firewall, který zastupuje klienta. Klient si je vědom přítomnosti firewallu.
- Transparentní proxy – Klient navazuje spojení přímo se serverem, ale firewall zasahuje do přenášených zpráv. Klient si není (nemusí být) vědom přítomnosti firewallu.

Struktura typického firewallu je zobrazena na obrázku 7.10. Jednotlivé části představují řetězce (bloky) pravidel, kterými jsou:



Obrázek 7.10: Části firewallu.

- INPUT – pravidla pro příchozí zprávy.
- OUTPUT – pravidla pro odchozí zprávy.
- FORWARD – pravidla pro přeposílání zpráv mezi síťovými rozhraními.
- PREROUTING – pravidla pro úpravu záhlaví příchozích zpráv (například cílová IP adresa a port). Jedná se o tzv. DNAT (Destination Network Address Translation).
- POSTROUTING – pravidla pro úpravu záhlaví odchozích zpráv. Jedná se o tzv. maškarádu (masquerade) neboli SNAT (Source Network Address Translation).

V jednotlivých řetězcích záleží na pořadí pravidel, protože jsou procházeny sekvenčně. Na konci celého řetězce je obvykle uvedeno výchozí pravidlo, které zajistí naložení se zprávami neshodujícími se s žádným pravidlem. Výchozí pravidlo všechny zprávy buď povolí (propustí) nebo naopak zakáže (zahodí).

7.5 Příklady na síťový systém

Příklady zahrnují kombinaci znalostí o síťové části, nástrojů pro vzdálenou práci a bezpečnosti operačních systémů.

Nejdříve bude popsán příklad netradičního testu dostupnosti stanice. Zde bude využit jiný nástroj, než je standardní příkaz `ping`. Bude využito zaslání segmentu s nastaveným příznakem SYN (žádost o navázání TCP spojení) na zvolený port. Tento způsob zjištění dostupnosti stanice lze v praxi využít například pokud jsou zprávy ICMP protokolu blokovány firewallem.

Dále bude popsán příklad získání informací o síťových službách. Jedná se o rozšíření předchozího příkladu s tím rozdílem, že testovány budou služby na jednotlivých portech. Pomocí tzv. skenování portů lze získat přehled o všech dostupných službách. V příkladu bude také popsáno zjištění konkrétní verze služeb na serveru.

Mimo informací o jednotlivých službách lze také odhadnout typ operačního systému. K tomu lze využít implementačních odlišností protokolové sady TCP/IP (tzv. TCP/IP otisku). Nakonec bude ukázáno sledování průběhu komunikace nezabezpečeného protokolu TELNET.

7.5.1 Netradiční test dostupnosti

Klasicky je dostupnost stanice testována pomocí protokolu ICMP a to kombinace jeho zpráv ECHO REQUEST a ECHO REPLY. Tento test realizuje program `ping`. Často jsou však tyto zprávy blokovány firewallem a nelze tak stav stanice zjistit. Alternativním způsobem je navázání spojení se zvoleným číslem portu. Toho lze dosáhnout zasláním segmentu s nastaveným příznakem SYN (žádost o navázání spojení). Pokud stanice odpoví (kladně či záporně) je on-line. Pokud stanice neodpoví, je buď nedostupná, nebo je zvolený port blokován firewallem.

K testu dostupnosti stanice lze využít program `nmap` (Network Mapper)[9]. Ve výpisu 7.9 je parametrem `sn` zakázán test portů (viz dále) a parametrem `PS22` je zaslán segment s příznakem SYN na port 22. Z výpisu je patrné, že stanice je dostupná[10].

```
[]$ nmap -sn -PS22 localhost
Nmap scan report for localhost (127.0.0.1)
Host is up.
rDNS record for 127.0.0.1: localhost.localdomain
Nmap done: 1 IP address (1 host up) scanned in 0.00 seconds
```

Výpis kódu 7.9: Netradiční test dostupnosti stanice.

[9] `http://nmap.org/`

[10] Z výukových důvodů a snadno opakovatelných postupů testujeme místní stanici.

7.5.2 Získání informací o službách

Předchozí příklad testu dostupnosti stanice lze rozšířit pro detekci služeb na jednotlivých portech stanice. Aplikace a služby operačního systému pracují s adresami, které se nazývají porty. Po spuštění aplikace, která realizuje server, dojde k obsazení příslušného portu vzniklým procesem. Server pak na této adrese naslouchá na příchozí spojení.

K získání informací o službách operačního systému lze opět využít program `nmap`. Program ke zjištění informací o službách využívá několik způsobů. Mezi ně například patří upravené navázání spojení pomocí protokolu TCP. Uvažme soket svázaný s daným portem, který je ve stavu naslouchání pro příchozí spojení (LISTEN). Pokud přijde na tento soket žádost o spojení, tak soket odpoví potvrzením navázání prvního jednosměrného spojení. Programem `nmap` je tedy zaslán TCP segment s nastaveným příznakem SYN, což je první segment pro navázání spojení se serverem. Odpověď v podobě segmentu s nastavenými příznaky SYN/ACK značí, že port je otevřený. Tímto dojde k částečnému navázání spojení (pouze jedno jednosměrné spojení). Následně je ukončeno navazování spojení pomocí zaslání segmentu s příznakem RST. Alternativou je zaslání segmentu s příznakem FIN, který slouží pro ukončení spojení. Principem této detekce dostupnosti služeb je, že některé operační systémy odpovídají na uzavřené porty jinak než na porty otevřené.

Příklad detekce otevřených portů a jejich služeb je zobrazen ve výpisu 7.10.

```
[]$ nmap localhost
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000040s latency).
rDNS record for 127.0.0.1: localhost.localdomain
Not shown: 995 closed ports
PORT     STATE SERVICE
22/tcp   open  ssh
25/tcp   open  smtp
5900/tcp open  vnc

Nmap done: 1 IP address (1 host up) scanned in 0.09 seconds
```

Výpis kódu 7.10: Přehled otevřených portů a svázaných služeb operačního systému.

Z výpisu je patrno, že na stanici běží několik služeb na portech 22, 25, a 5900. Služba SSH slouží pro vzdálené textové připojení. SMTP (Simple Mail Transfer Protocol) je služba pro přenos emailů. VNC (Virtual Network Computing) umožňuje připojení pomocí vzdálené plochy.

Dále lze také získat verze jednotlivých služeb. Jaký je k tomu důvod? Pokud známe verzi dané služby, tak lze v databázích dohledat, jaké jsou její bezpečnostní chyby. Detekcí verze lze tyto chyby odhalit a upozornit na ně, než je někdo zneužije k útoku. Ukázka zjištění verze služeb je zobrazena ve výpisu 7.11.

```
[]$ nmap -A localhost
PORT     STATE SERVICE VERSION
22/tcp   open  ssh   OpenSSH 5.3 (protocol 2.0)
| ssh-hostkey: 1024 c5:aa:...:46:1c (DSA)
|_2048 45:f6:7d:b0:39:00:d6:... (RSA)
```

```
25/tcp   open  smtp    Postfix smtpd
5900/tcp open  vnc     VNC (protocol 3.7)
```

Výpis kódu 7.11: Zjištění verzí provozovaných služeb.

Pro službu SSH se podařilo odhalit, že se jedná o implementaci OpenSSH ve verzi 5.3. Ve výpisu můžeme zjistit i veřejný klíč stroje. Tento klíč slouží k detekci útoku, kdy se nějaká stanice vydává za pravý server. Tento útok lze vytušit pomocí změny tohoto klíče. Další informací ve výpisu je verze protokolu RFB (Remote Frame Buffer) 3.7, který využívá program VNC pro realizaci vzdálené plochy.

Pro zjištění typu operačního systému může být využita analýza implementačních odlišností protokolové sady TCP/IP (tzv. TCP/IP otisk). Způsob zjištění tohoto otisku spočívá ve sběru informací o protokolové sadě TCP/IP pomocí komunikace s nastavenými specifickými parametry. Využívá se té skutečnosti, že parametry protokolové sady TCP/IP se liší podle toho, jak jsou implementovány v daném operačním systému. Různé operační systémy (a někdy i jejich verze) se odlišují v této implementaci, a tudíž poskytují různé odpovědi na specifické dotazy. Například se vyhodnocuje počáteční velikost paketu, hodnota TTL, velikost okna, velikost MTU atd. Vrácené hodnoty jsou porovnány s databází hodnot pro různé operační systémy, a tak je proveden vlastní odhad.

Ve výpisu 7.12 je vidět, že program `nmap` nedokázal přesně odhalit operační systém Linux distribuce CentOS, který je provozován na dotazovaném serveru. Provedl pouze odhad typu operačního systému s pravděpodobností odhadu v závorce. Z odhadu je patrné, že správně byl odhadnut OS Linux (96 %). Dále z výpisu je patrno, že byl proveden i odhad typu zařízení.

```
[]$ nmap -O -osscan-guess localhost
Device type: WAP|general purpose|webcam|firewall|specialized|storage-
    misc
OS guesses: Netgear DG834G WAP (96%), Linux 2.6.19 - 2.6.36 (96%), Linux
    2.6.22 - 2.6.23 (93%), ...
No exact OS matches for host (If you know what OS is running on it, see
    http://nmap.org/submit/ ).
```

Výpis kódu 7.12: Zjištění typu operačního systému, varianta A.

Zajímavostí je i možnost nahlásit typ testovaného systému na adrese `http://nmap.org/submit/`, a tak pomoci vylepšit funkci programu `nmap` (poslední řádek ve výpisu).

Ve výpisu 7.13 je zobrazena část TCP/IP otisku. Z těchto informací lze vyčíst, že se jedná o operační systém `i386-redhat-linux-gnu`, což odpovídá skutečnosti[11]. Další zobrazené hodnoty jsou implementační parametry TCP/IP.

```
[]$ nmap -O -v localhost
TCP/IP fingerprint:
OS:SCAN(V=5.51%D=10/9%OT=22%CT=1%CU=39368%PV=...
OS:i386-redhat-linux-gnu)SEQ(SP=F7%GCD=1%...
OS:=F7%GCD=2%ISR=10F%TI=Z%CI=Z%II=I%TS=A)OPS(O1=...
```

Výpis kódu 7.13: Zjištění typu operačního systému, varianta B.

[11] Použitá distribuce CentOS je binárně shodná s distribucí RedHat – je jejím klonem.

7.5.3 Získání obsahu komunikace

Data budou zachycena pomocí nástroje v textovém režimu. Práce v textovém režimu má své opodstatnění, protože v reálné síti je monitorování provozu prováděno na síťových prvcích, které nemusí mít k dispozici grafické rozhraní, tak jako koncové stanice. Pro reálné využití je potřeba mít přístup na síťové zařízení, přes které probíhá komunikace. Na tomto zařízení pak spustit program pro zachytávání komunikace a výsledky odesílat na sběrnou stanici. Ideálním místem je výchozí brána, přes kterou je směrována veškerá komunikace mimo lokální síť. Lze tak nalézt stanice/uživatele, kteří tvoří slabé místo sítě.

Server `telnetd`[12] je klasicky spouštěn pomocí superdémona, např. `xinetd`. Postup spuštění je: `cd /etc/xinetd.d/; vi telnet`, nastavení hodnoty *disable=no*, `/etc/init.d/xinetd restart`. Pro zachytávání dat lze využít program `tcpflow`. Na vzdáleném serveru budou provedeny tyto operace – vytvoření adresáře a v něm souboru, následně soubor bude smazán, tedy `mkdir pokus`; `touch pokus/pokus.txt`; `rm pokus/pokus.txt`.

Výsledek je zachycen ve výpisu 7.14. Parametrem `-i` specifikujeme rozhraní pro zachytávání a portem 23 určujeme protokol TELNET. Zachycená komunikace je uložena v souboru s názvem IP adresy a použitého portu pro obě strany komunikace. Část názvu `X.00023` označuje spojení na straně serveru. Část názvu `X.59576` označuje spojení na straně klienta. Zobrazením obsahu souboru lze zjistit, jaké akce uživatel *bsos* prováděl na serveru. Z výpisu lze lehce rozpoznat, že zachycené příkazy odpovídají skutečným akcím na serveru.

```
[]$ tcpflow -i lo -s port 23
...
[]$ more 127.000.000.001.00023-127.000.000.001.59576
Last login: Thu Oct 11 17:49:26 from localhost
mkdir pokus
touch pokus/pokus.txt
rm pokus/pokus.txt
exit
odhlaseni
```

Výpis kódu 7.14: Zachycená komunikace TELNET.

[12] Pro instalaci serveru TELNET je nutno použít tento příkaz `yum install telnet-server`.

8 ZÁVĚREM

Předmět měl za cíl poskytnout obecné základy síťových operačních systémů, které lze dále rozvinout pro konkrétní oblasti použití. Mým záměrem bylo skripta napsat tak, aby jednotlivé části do sebe zapadaly v přeneseném slova smyslu jako „ozubená kola“ a dávaly tak celistvý obraz o principech a činnosti. Při výkladu teorie jsem použil příklady, aby byla problematika lépe pochopitelná.

Rozumět operačním systémům a věcem s tím spojených – programování, bezpečnost, ale třeba i reverzní inženýrství, není o tom, že víte na co „kliknout“. Přístup klikače a znalost obsluhy konkretních koncových aplikací je krátkozraká a na jedno použití (jsou – nebudou; budou jiné, a to brzy). Znalost principů má dlouhodobou hodnotu, má univerzální využití, a to bylo mou snahou při tvoření tohoto předmětu. Praktická část byla zaměřena na obecné nástroje, které lze využít pro různé účely.

Nakonec jedna citace „*Today's learners need to know not only basic knowledge (i.e., factual and conceptual) level, but also advanced skills that allow them to face a world that is continually changing (i.e., procedural and metacognitive knowledge).*“ [38].

REFERENCE

[1] Raymond E. S. The Art of UNIX Programming. Addison-Wesley Professional Computing Series, Massachusetts, 2003. First Edition, ISBN-13: 978-0131429017.

[2] Thompson K., Ritchie D. M. The UNIX Time–Sharing System. Communications of the ACM, 17(8):365–375, July 1974.

[3] Ritchie D. M. The Development of the C Language. ACM SIGPLAN Notices, 28(3):201–208, March 1993.

[4] The Open Group. The Single UNIX® Specification. The Authorized Guide to Version 3, 2002.

[5] Garfinkel S., Spafford G. Bezpečnost v UNIXu a Internetu v praxi. Computer Press, Praha, 1998. Vydání první.

[6] Macur, J. X Window, grafické rozhraní operačního systému UNIX. SCIENCE, Veletiny, 1994. Vydání první, ISBN: 80-901475-1-8.

[7] Bach, M. J. Principy operačního systému UNIX. Softwarové Aplikace a Systémy, Praha, 1993. Vydání první, ISBN: 80-902612-5-6.

[8] Barrett, D. J., Silverman R. E. SSH, The Secure Shell: The Definitive Guide. O'Reilly & Associates, Inc., Sebastopol, 2001. First Edition ,ISBN-13: 978-0596008956.

[9] Koudelka, P. Historie operačních systémů. [online]. 2013 [cit. 11. 4. 2013]. Dostupné z http://airborn.webz.cz/histos.html.

[10] Schade, O. Microsofts Evolution of Technology. [on-line]. 2002 [cit. 7. 12. 2011]. Dostupné z: http://internet.ls-la.net/.

[11] Tanenbaum, S. Modern Operating Systems. Prentice Hall PTR., 2007. Third Edition, ISBN-13: 978-0136006633.

[12] Stevens, W. R. UNIX Network Programming, Networking APIs: Sockets and XTI, Volume 1. Prentice Hall PTR., 2003. Third Edition, ISBN-13: 978-0131411555.

[13] Kolektiv autorů. Linux Dokumentační projekt. Computer Press, Brno, 2003. Třetí vydání, ISBN: 80-7226-761-2.

[14] Dostálek, L., Kabelová, A. Velký průvodce protokoly TCP/IP a systémem DNS. Computer Press, Brno, 2008. Páté vydání, ISBN: 978-80-251-2236-5.

[15] Dostálek, L. a kol. Velký průvodce protokoly TCP/IP - bezpečnost. Computer Press, Brno, 2001. První vydání, ISBN 80-7226-513-X.

[16] Silberschatz, A., Galvin, P., Gagne, G. Operating systems concepts. John Wiley, 2008. Eight edition, ISBN-13: 978-0470128725.

[17] Skočovský, L. Principy a problémy operačního systému UNIX. SCIENCE, Veletiny 212, 1993. Vydání první, ISBN: 80-901475-0-X.

[18] Harrisová S,. Harper A., Eagle, Ch., et al. Grada Publishing, 2008. ISBN: 978-80-247-1346-5.

[19] Toxen, B. Bezpečnost v Linuxu, Computer Press, 2003. ISBN: 80-7226-716-7.

[20] DeLaet, G., Sxhauwers, G. Network Security Fundamentals, Cisco Press, 2004. First edition, ISBN-13: 978-1587051678.

[21] Timeline of Computer History. [on-line]. 2006 [cit. 7. 12. 2011]. Dostupné z http://www.computerhistory.org/timeline/.

[22] Russinovich, M., Solomon, A. Vnitřní architektura Microsoft Windows. Computer Press, 2006. ISBN: 802-5112-66-7.

[23] TFTP: Trivial File Transfer Protocol. [on-line]. 2011 [cit. 7. 12. 2011]. Dostupné z http://www.javvin.com/protocolTFTP.html.

[24] AT&T Laboratories, Cambridge. VNC - How it works. [on-line]. 1999 [cit. 7. 12. 2011]. Dostupné z: http://virtuallab.tu-freiberg.de/p2p/p2p/vnc/ug/howitworks.html.

[25] Cisco. CCNA Exploration 4 – Accessing the WAN [on-line]. 2011 [cit. 7. 12. 2011].

[26] ITU-T. Recommendation X.200: Information technology - Open Systems Interconnection - Basic Reference Model: The basic model [CD-ROM]. 1994.

[27] Yukun, L., Guo, L., Yong, Y. UNIX Operating System - The Development Tutorial via UNIX Kernel Services. Springer, London, 2011. ISBN: 978-3-642-20431-9.

[28] Gancarz, M. The UNIX Philosophy. Digital Press, 1994. ISBN: 978-1555581237.

[29] Chu, W., Opderbeck, H. The page fault frequency replacement algorithm. Proceeding of AFIPS'72 (Fall, part I). Fall joint computer conference. ACM New York, 1972.

[30] McHoes, A., Flynn, I. Understanding Operating Systems. Course technology – Cengage learning, 2011. International Edition. ISBN: 978-0-538-47004-9.

[31] Horman, N. Understanding Virtual Memory in Red Hat Enterprise Linux. Red Hat, 2005. White paper, version 0.1.

[32] Arpaci-Dusseau, R. Arpaci-Dusseau, A. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, LLC, 2016. ISBN: n/a.

[33] Braden, R. Requirements for Internet Hosts – Communication Layers. Internet Engineering Task Force, 1989. Request for Comments: 1122.

[34] Touch, J. Recommendations on Using Assigned Transport Port Numbers. Internet Engineering Task Force, 2015. Request for Comments: 7605.

[35] Linux Foundation. Filesystem Hierarchy Standard. LSB Workgroup, The Linux Foundation, 2015. Version 3.0.

[36] Chiramana, S. Python Functools – lru_cache(). [on-line]. 2020 [cit. 16. 1. 2021]. Dostupné z: https://www.geeksforgeeks.org/python-functools-lru_cache/.

[37] Cotton, M. et al. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. Internet Engineering Task Force, 2011. Request for Comments: 6335.

[38] Yousef, F. and Sumner, T. Reflections on the last decade of MOOC research. Computer Applications in Engineering Education, 2021, roč. 29, č. 4, s. 648-665.