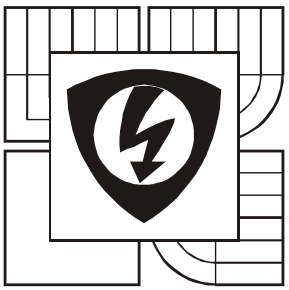
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



Síťové operační systémy

Garant předmětu:
prof. Dan Komosný

Autoři textu:
Dan Komosný a kolektiv

BRNO 2023

Obsah

1 INFORMACE O PŘEDMĚTU

Cílem předmětu je poskytnout obecný pohled na problematiku operačních systémů. Předmět se zabývá síťovými operačními systémy[1].

První část předmětu je zaměřena na obecnou teorii. Je zde probírána architektura, procesy, paměti a souborové systémy. V druhé části je zahrnut popis síťové části a bezpečnosti. Pro snazší pochopení je probíraná teorie podpořena příklady.

Na těchto skriptech se podíleli i další autoři, zejména Michal Soumar, Patrik Morávek a Jiří Balej. K úrovni skript také přispěli studenti díky připomínkám a námětům. Skripta jsou průběžně aktualizována a proto vždy pracujte s aktuální verzí.

[1] Pojem *síťový operační systém* není jednoznačně definován. Převládá přístup, že síťový operační systém je takový, který pracuje na serveru a poskytuje služby (nesíťovým) operačním systémům.

2 ZÁKLADNÍ KONCEPT

2.1 Funkce a vstup/výstup

Operační systém poskytuje dvě základní funkce, které jsou **rozšířený automat** a **správa zdrojů** [11].

Rozšířený automat (extended machine) umožňuje jednodušší obsluhu než přímá práce s hardware. Operační systém poskytuje množinu služeb, ke kterým programy přistupují pomocí **systémových volání**. Jedná se například o čtení dat ze souboru. Programátor nemusí znát detaily ohledně konkrétní pozice dat na paměťovém médiu. Je také skryta obsluha přerušení, správa paměti atd. Rozšířený automat používá pohled „shora dolů“.

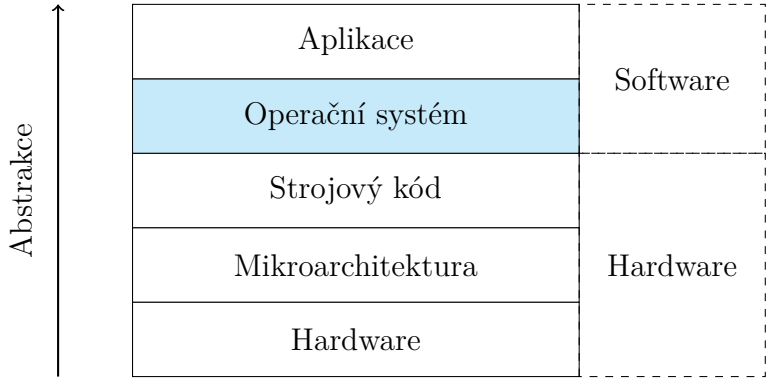
Správa zdrojů (resource manager). Počítačové systémy zahrnují více zdrojů, jako je například procesor, paměť, časovače, paměťová média, síťová rozhraní atd. Těmito zdroji není jen hardware, ale také informace (soubory, databáze atd.). Úkolem operačního systému je poskytnout programům přístup k těmto zdrojům. Operační systém udržuje informaci o tom, který proces[1] využívá který zdroj. Operační systém řeší konflikty při (pokusu) využívání jednoho zdroje současně více procesy. Zdroje jsou používány v časové a prostorové rovině. V časové rovině se procesy o zdroj postupně střídají. Příkladem je střídání o procesor. V prostorové rovině procesy používají část daného zdroje současně. Příkladem je hlavní paměť (RAM), která je rozdělena na dílčí bloky. Správa zdrojů používá pohled „zdola nahoru“.

Pozice operačního systému ve výpočetním prostředku je zobrazena na obrázku 2.1. Mikroarchitektura poskytuje základní prvky, jako jsou registry, sčítače, násobiče a čítače. Strojový kód je sled bitů, kterými se tyto prvky ovládají. Strojový kód je záznam instrukcí a operandů **instrukční sady architektury** (ISA – Instruction Set Architecture). Instrukční sada typicky obsahuje 50 až 300 instrukcí. Instrukce například provádějí přesun dat, aritmetické operace a porovnání hodnot. Příklad sečtení hodnot v registrech a strojového kódu je zobrazen ve výpisu 2.1.

```
1 #Soucet hodnot v registrech r1=r2+r3 (nazvy registru a bity ukazkove)
2 add r3 r2 r1 --> 0001 0111 0100 1011
3
4 #Strojovy kod - sled instrukci a operandu (bity ukazkove)
5 cmp  ax   0x5  mov  ax   0x8
6 0110 0011 1000 1100 0011 1000
```

Výpis kódu 2.1: Příklad sečtení hodnot v registrech a strojového kódu.

[1] Proces je instance spuštěného programu.



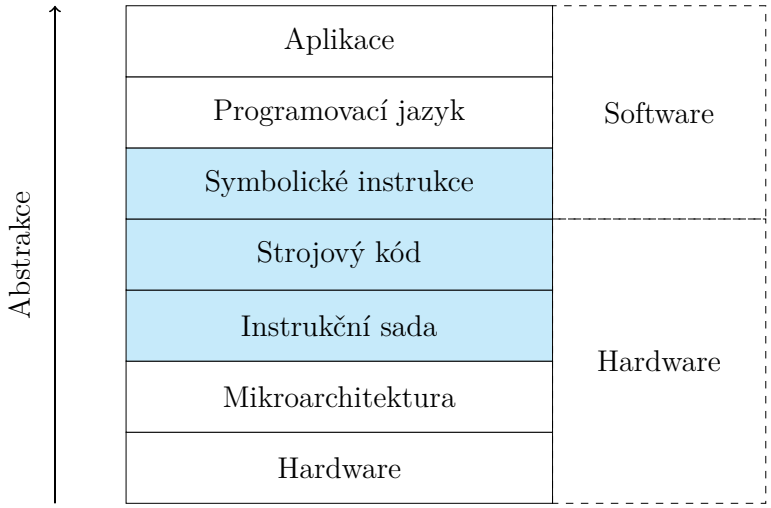
Obrázek 2.1: Pozice operačního systému.

Pro přímé programování se používá jazyk **symbolických instrukcí a adres** (assembly language). V tomto jazyce jsou instrukce a registry reprezentovány symboly. Symboly podle svého názvu označují, co instrukce provádí (např. `mov` – přesun dat), či které místo v paměti se adresuje (např. `ax` – název registru). Příklad je uveden ve výpisu 2.2, který provádí porovnání a odečet hodnoty. Překlad symbolů na strojový kód provádí program **asembler**.

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

Výpis kódu 2.2: Příklad jazyka symbolických instrukcí.

Pozice symbolických instrukcí, strojového kódu a instrukční sady je zobrazena na obrázku 2.2.



Obrázek 2.2: Symbolické instrukce a strojový kód.

Instrukční sada slouží k ovládání dané architektury systému (např. IA-32, x86-64, ARM). Stejná architektura může být realizovaná různými procesory. Například architekturu IA-32 realizovaly procesory 386, 486 a Pentium. Jak pracovat s danou architekturou popisuje **rozhraní ABI** (Application Binary Interface). S tímto rozhraním pracují kompilátory zdrojových kódů a ABI definuje strukturu přeložených programů, tj. jejich binární podobu. Definováno je například, kam mají být uloženy vstupní parametry a kde je návratová hodnota (ve kterých registrech). Naproti tomu **rozhraní API** (Application Programming Interface) je popis pro programátora, jak využívat externí funkce ve zdrojovém kódu. API je popsáno pomocí programovacího jazyka (C++, Python) a je nezávislé na architektuře systému.

Základní vstup a výstup je realizován programem **BIOS** (Basic Input-Output System). BIOS z paměťového média načte zavaděč, který spustí operační systém. Operační systém následně pracuje s hardware přímo, jelikož je to rychlejší než pomocí programu BIOS. Program BIOS umožňuje zobrazit informace o výpočetním prostředku. K tomu lze využít příkaz `dmidecode` (dmi – Desktop Management Interface). Výpis 2.3 zobrazuje velikost paměti ROM, ve které je BIOS umístěn. Funkce programu BIOS jsou dostupné pomocí přerušení. Příklad přerušení je uveden na konci výpisu.

```
[]# 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)
```

Výpis kódu 2.3: Zjištění informací o výpočetním prosředku a příklad přerušení.

Jedním ze základních vstupů a výstupů je čtení dat ze souboru. Při přímém přístupu je do registrů uložen příkaz k provedení (čtení/zápis) a jeho parametry. Tyto parametry pro pevný disk jsou:

- počáteční sektor,
- počet sektorů,
- cylindr,
- hlava plotny,
- identifikátor disku,
- kam uložit načtená data (adresa paměti).

Následně je zavolána funkce programu BIOS pro práci s daty na paměťovém úložišti. Funkce je zavolána pomocí přerušení (č. 0x13). Program BIOS si z registrů přečte kód příkazu a jeho parametry. Výsledkem přerušení jsou načtená data v paměti a návratová

hodnota. Návratová hodnota značí úspěch, nebo nese informaci o chybě. Příklady neúspěchu jsou neplatný příkaz, sektor nenalezen, detekován špatný sektor atd. Uvedený postup je ve zjednodušené podobě uveden ve výpisu 2.4.

```
1  mov ah,0x02    #kod prikazu - cist data z disku
2  mov cl,0x0     #prvni sektor
3  mov al,0x2     #kolik sektoru
4  mov ch,0x0     #cylindr
5  mov dh,0x0     #hlava disku
6  mov dl,0x80    #prvni pevny disk
7  mov bx,0x5000  #kam se maji data nacist (adresa pameti)
8  int 0x13       #preruseni BIOS c. 0x13
9
10 Navratova hodnota je v registru ah (0 znaci uspech)
```

Výpis kódu 2.4: Příklad čtení dat ze souboru bez operačního systému.

Operační systém provádí tyto náležitosti automaticky. Uživatel či programátor tak má o starost méně.

2.2 Historie

Stěžejní kroky při historickém vývoji operačních systémů byly:

- abstrakce systémových prostředků,
- přenositelnost mezi různými hardwarovými platformami,
- interaktivní režim,
- práce více uživatelů současně.

Počítače první generace, které vznikaly krátce po 2. světové válce, operační systém neobsahovaly. Programy se psaly v jazyce symbolických instrukcí a obsluha musela vědět, na kterých vstupech a výstupech může pracovat (registry). Program byl uložen na propojovacích deskách, později se používaly magnetické pásky. Následně bylo vyvinuto několik programovacích jazyků[2]. Programy byly kompilovány (přeloženy do strojového jazyka) na příslušný hardware. Vstupy/výstupy (registry) byly stále adresovány ručně. Bylo tedy potřeba vyvinout program, který by vnitřní funkce systému prováděl automaticky. Začaly vznikat první operační systémy, které prováděly abstrakci hardware a spravovaly zdroje.

Vznikly různé typy operačních systémů, včetně síťových. Dominantními síťovými operačními systémy se staly UNIX a GNU/Linux. UNIX představuje označení pro celou rodinu operačních systémů. Operační systém lze označit jako UNIX po splnění sady testů. Pro popis OS UNIX lze použít citaci, jejímž autorem je 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.*“[3].

[2] Například Fortran a COBOL.

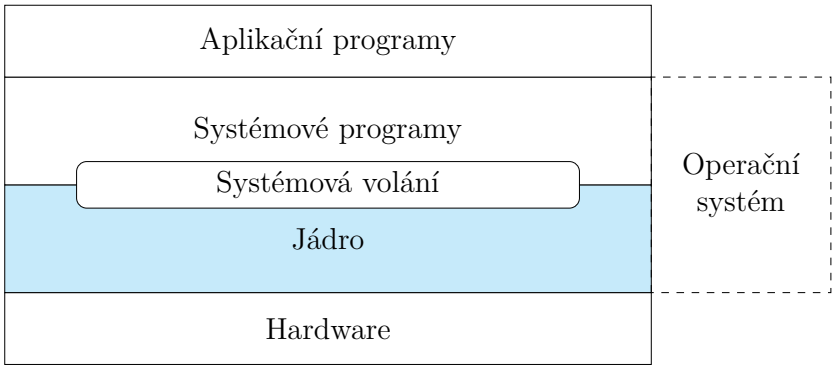
[3] Pište programy, které dělají jen jednu věc, ale dělají ji dobře. Pište programy, které pracují dohromady. Pište programy, které pracují s textovými proudy, protože ty představují univerzální rozhraní.

Linux je volně šiřitelné systémové jádro. Linuxová distribuce obsahuje jádro a další software. Tento software je typicky distribuován pod svobodnou licencí GNU. Správné označení operačního systému je tedy GNU/Linux, což vychází z použitého jádra Linux a dalšího software. Běžně se však používá zkrácené označení Linux, které také budeme používat v tomto textu.

3 ARCHITEKTURA

3.1 Struktura systému

Operační systém obsahuje **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 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.

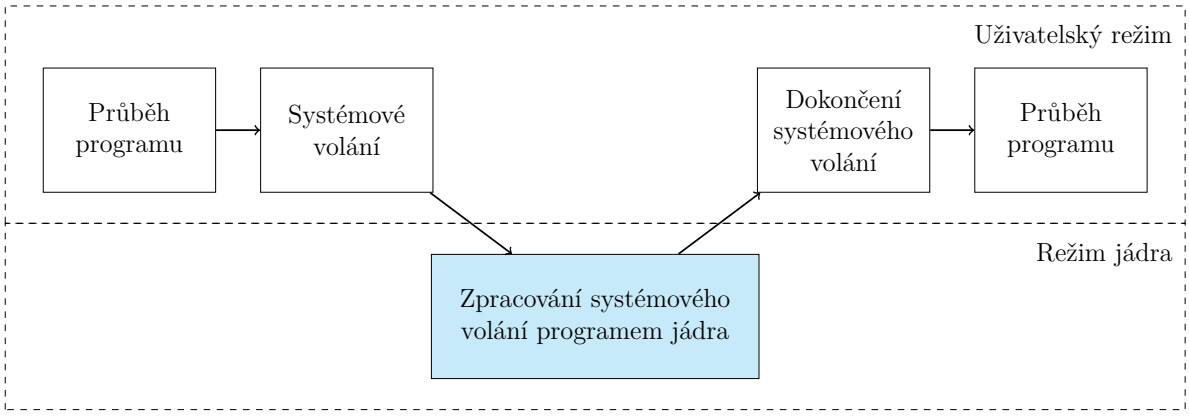


Obrázek 3.1: Struktura operačního systému.

Operační systém pracuje ve dvou základních režimech, kterými jsou **režim jádra** a **uživatelský režim**[1]. Režim jádra označujeme také jako privilegovaný/systémový režim nebo supervizor. Uživatelský režim označujeme také jako neprivilegovaný. Jádro a vybrané systémové programy pracují v privilegovaném režimu bez omezení. Programy v uživatelském režimu jsou omezovány tím, že procesor jim neumožňuje provést určité operace. Pokud program v neprivilegovaném režimu chce provést chráněnou operaci, musí „požádat“ program v privilegovaném režimu. Ten provede kontrolu 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, zakázání přerušení, práce s I/O zařízeními. Obecně jsou to operace, které vyžadují koordinaci nebo které mohou ohrozit činnost OS, například chybou v programu. Chyba v jádře/systémovém programu může vyřadit celý systém. Chyba v uživatelském režimu ovlivní chod daného programu.

[1] Toto rozdělení není platné pro všechny systémy. Například dedikované systémy nemusí režimy rozlišovat a mají pouze privilegovaný režim.

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 (přerušení je potřeba 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, tj. do kterých registrů. Tato specifikace je součástí binárního rozhraní ABI pro danou architekturu. Bližší popis ABI pro jednotlivé architektury lze například najít v manuálových stránkách systémového volání *syscall* (`man syscall`).

I jednoduchý program v uživatelském režimu, kde se ve zdrojovém kódu volá pouze jedno systémové volání, ve skutečnosti využívá 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, 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 Monolitický systém a modulární jádro

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 objektů 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.

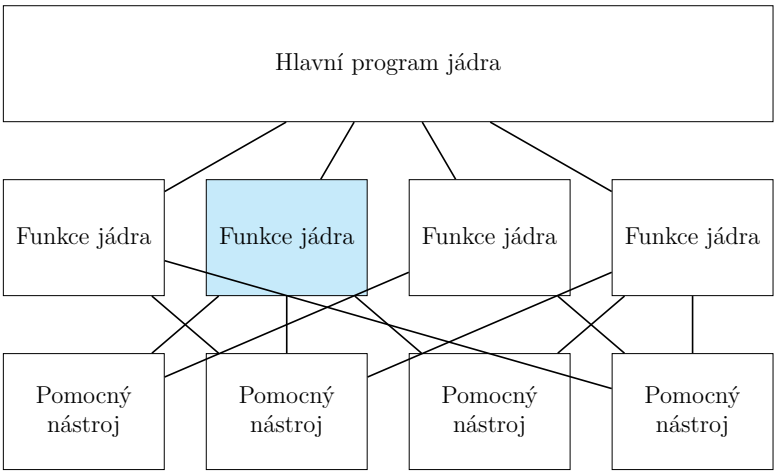
Navzdory tomu, že z globálního pohledu není zřejmá nějaká struktura, lze strukturu monolitického jádra popsat následovně:

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

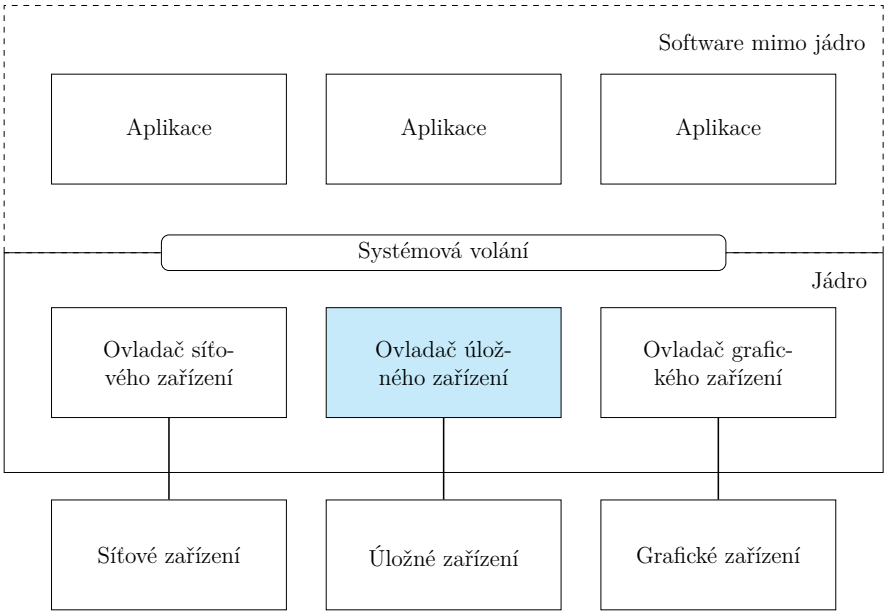
Hlavní program spouští funkci pro obsluhu systémového volání/přerušení. Každé syst. volání/přerušení obsluhuje jedna funkce. Úkolem pomocných nástrojů je provádět akce vyžadované několika funkcemi. Výsledný model je zobrazen na obrázku 3.3. Na obrázku jsou sice zobrazeny vrstvy, ale je nutno si jednotlivé 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. Další funkce jsou k dispozici v 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



Obrázek 3.3: Zjednodušená struktura monolitického jádra.



Obrázek 3.4: Monolitické jádro.

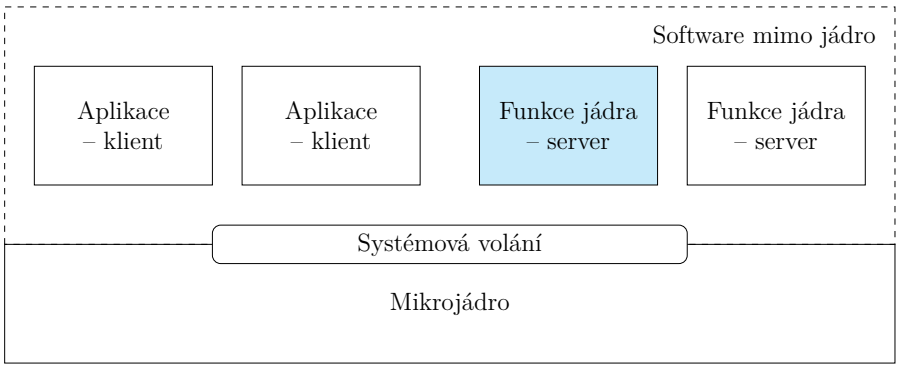
je také v menší velikosti jádra. Mezi moduly například patří ovladače zařízení a podpora souborových systémů.

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

3.1.2 Systém klient-server, mikrojádro a hybridní jádro

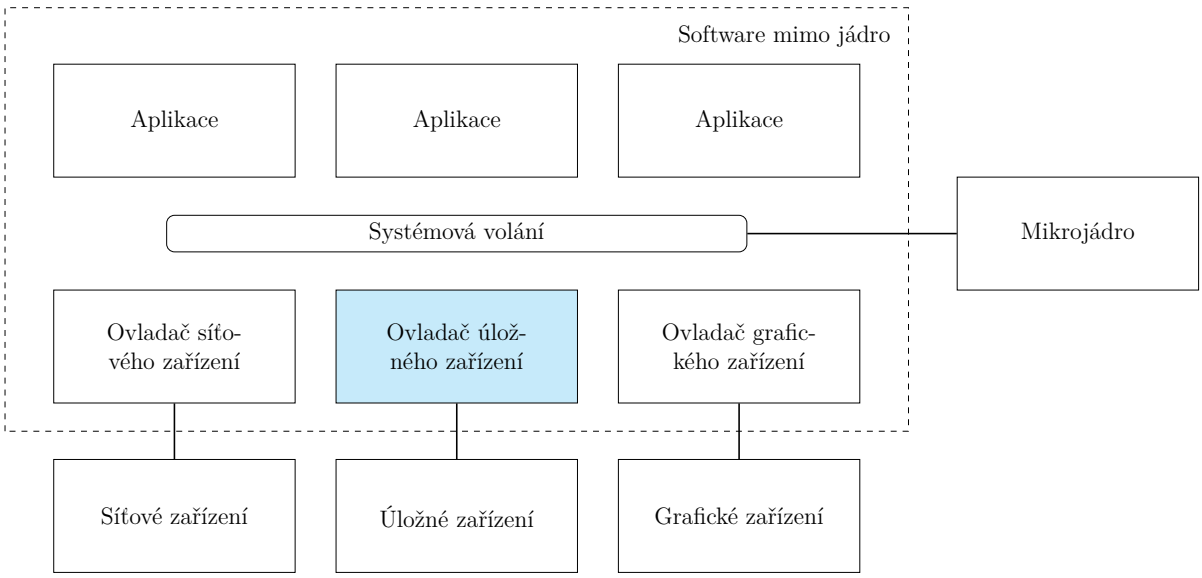
Systém klient-server vznikl přesunem funkcí z jádra do programů – serverů, které běží v uživatelském režimu [11]. Jádro programům přidělí privilegovaná práva pro přímý přístup do paměti DMA (Direct memory access). DMA umožňuje přímý přenos dat mezi

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 serveru způsobí 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 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

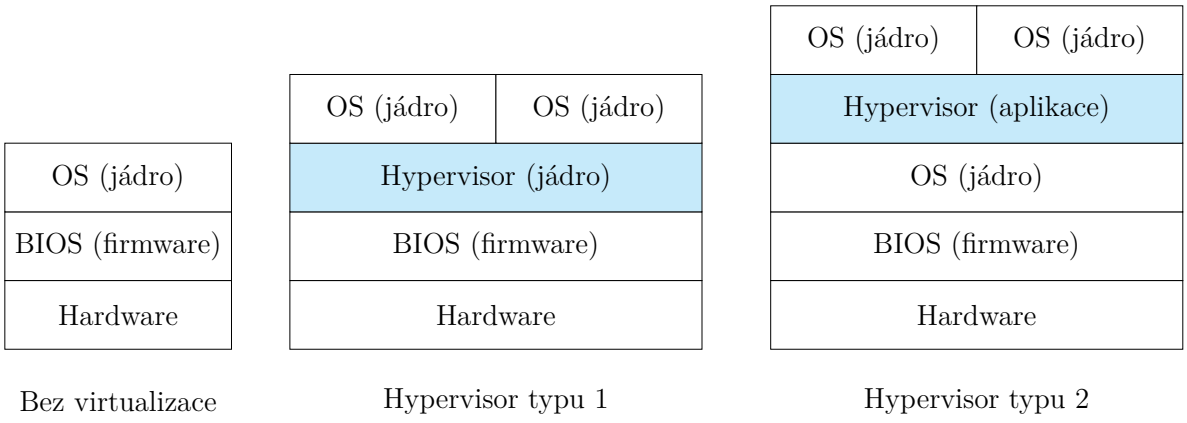
Principem tohoto přístupu je organizovat operační systém do **vrstev**. Vrstva je abstraktní objekt sestávající se z dat a operací manipulujících s těmito daty. Vrstva poskytuje služby pouze následující vyšší vrstvě a je implementována pouze pomocí funkcí její spodní vrstvy. Nejnižší vrstva je nad hardware 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 vrstvy může být využita pouze 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. Jelikož je provedena abstrakce hardware, lze stejné virtuální stroje provozovat nad různým 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 stroji 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 – zvyšuje spolehlivost a využití hardware. Pokud operační systém poskytuje službu na jednom hardware, je při selhání hardware tato služba nedostupná. Dochází také k plýtvání prostředků, jelikož hardware je využíván pouze při poskytování dané služby (web, email atd.). Ke správě virtuálních strojů provozovaných nad hypervisorem typu 1 slouží tzv. „management console“. Tento nástroj umožňuje automatický přesun virtuálního stroje na jiný hardware, a tím v krátkém čase provést obnovu služby. „Management console“ také dynamicky alokuje prostředky hardware pro virtuální stroje. Lze provést tzv. „over allocation“, tedy poskytnutí více prostředků než je k dispozici (např. RAM). 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 – zvyšuje bezpečnost pomocí oddělení virtuálních strojů. Přenos dat mezi virt. stroji lze řešit pomocí sdílení souborů na paměťovém úložišti nebo přes síť. Síťová komunikace může být pouze v privátní síti, ve které jsou propojeny

virt. stroje 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 LAN laboratoře[2].

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 objektů do jednoho spustitelného souboru. Každá procedura jádra je přímo „viditelná“ každé jiné.

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).

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

[2] První použití na fakultě pro laboratoř Cisco akademie.

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 inicializační souborový systém – soubor `initramfs*` (initial RAM file system). Tento systém obsahuje doplňkové součásti pro start operačního systému před tím, než dojde k připojení reálného kořenového souborového systému na paměťovém úložišti. Příkladem je poskytnutí záchranného terminálu pokud start OS selže nebo připojení šifrovaných oddílů.

Aktuálně používané moduly v jádře lze získat příkazem `lsmod`, jak je zobrazeno ve výpisu 3.4. Po startu jsou další moduly jádra k dispozici na připojeném paměťovém úložišti, typicky v adresáři `/lib/modules/<verze_jadra>/`.

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

Výpis kódu 3.4: Seznam modulů nahraných do jádra.

Informace o umístění souboru se specifickým modulem lze získat příkazem `modinfo <nazev_modulu>`. Ve výpisu 3.5 je zobrazen příklad umístění souboru s ovladačem pro bezdrátovou komunikaci bluetooth.

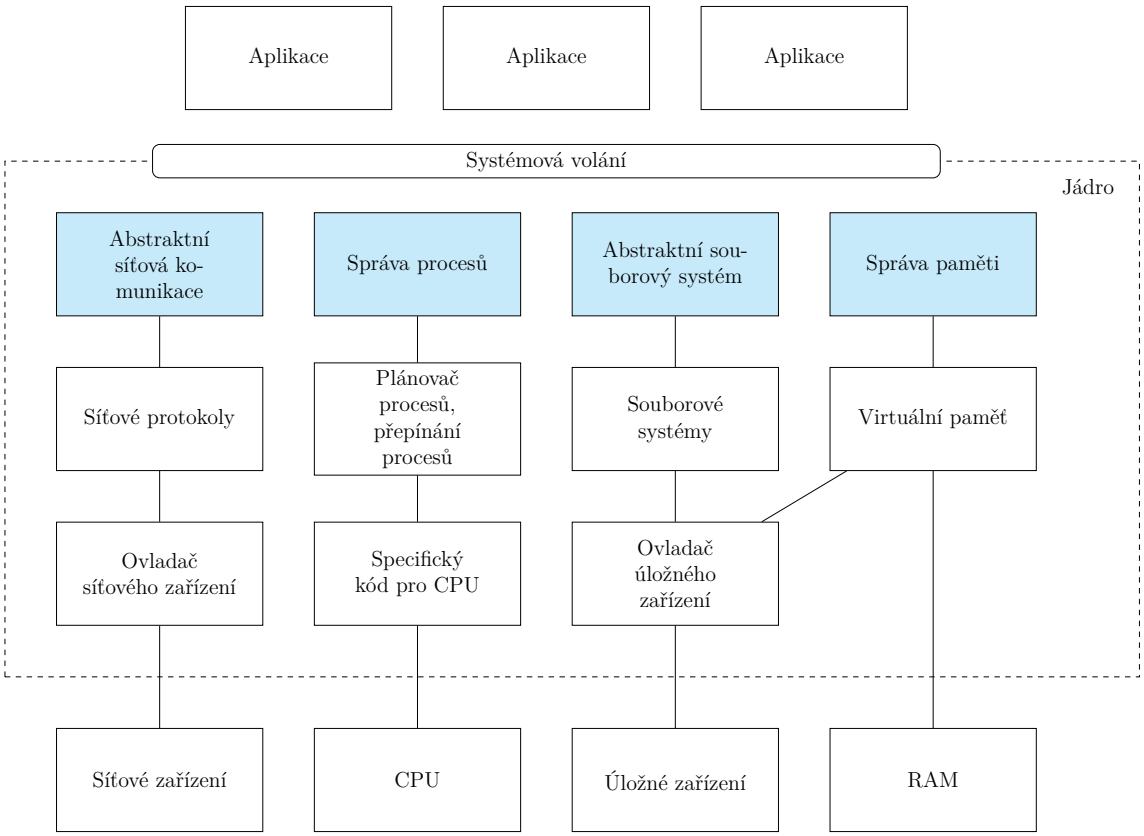
```
[]# modinfo bluetooth
filename: /lib/modules/../kernel/net/bluetooth/bluetooth.ko.xz
license: GPL
version: 2.22
description: Bluetooth Core ver 2.22
```

Výpis kódu 3.5: Soubor s modulem pro komunikaci bluetooth.

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 varianty jader.

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

Proč kompilovat jádro? Původně byl hlavní důvod kompilace jádra pro podporu systémové architektury. V dnešní době jsou k dispozici zkompilovaná jádra pro různé architektury, např. x86_64, ARM64 (aarch64) a IBM Power (ppc64le). Jádra distribucí jsou kompilována s univerzální konfigurací, aby pracovala co s nejširší hardwarovou výbavou. Pro běžné použití 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 s doplňkovými částmi (`initramfs`). 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 distribuci CentOS má velikost okolo 6 MiB – soubor `vmlinuz` v adresáři `/boot`, viz výpis 3.6. Znak *z* v názvu souboru značí, že se jedná o komprimovaný soubor (znaky *vm* označují použití konceptu virtuální paměti). Obraz virtuálního disku v RAM (`initramfs`) má velikost okolo 30 MiB. 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 3.6: Velikost jádra a obrazu virtuálního disku s moduly.

Co bude jádro obsahovat 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 vlastnostmi) bylo jádro zkompilováno. Přeložené moduly jádra jsou umístěny do adresáře `/lib/modules` (připojený souborový systém).

Instalace jádra typicky 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:

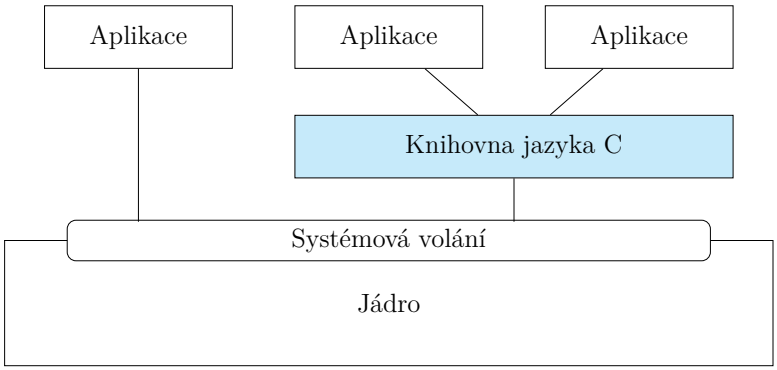
1. Stažení zdrojových kódů, například pomocí balíčku pro danou distribuci.
2. Konfigurace jádra.
3. Překlad jádra.
4. Instalace jádra do příslušných adresářů a úprava zavaděče systému pro výběr nového jádra.
5. Restart a zavedení nového jádra.

3.2.3 Systémové volání

Pro popis příkladu systémového volání si zopakujme, že:

- 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. Použití knihovny jako prostředníka přináší univerzálnost tím, že převádí systémová volání poskytované konkrétním jádrem na standardní funkce jazyka C a C++, které jsou nezávislé na jádře. V OS Linux je to knihovna *GNU C Library* (`glibc`). Tato knihovna podporuje různá jádra, včetně jader 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 jako systémový program.



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 pro čtení dat ze souboru [11]. Použita bude funkce s názvem *read* s těmito třemi parametry: soubor (deskriptor souboru), vyrovnávací paměť (buffer) a počet bajtů ke čtení, viz struktura parametrů funkce ve výpisu 3.7.

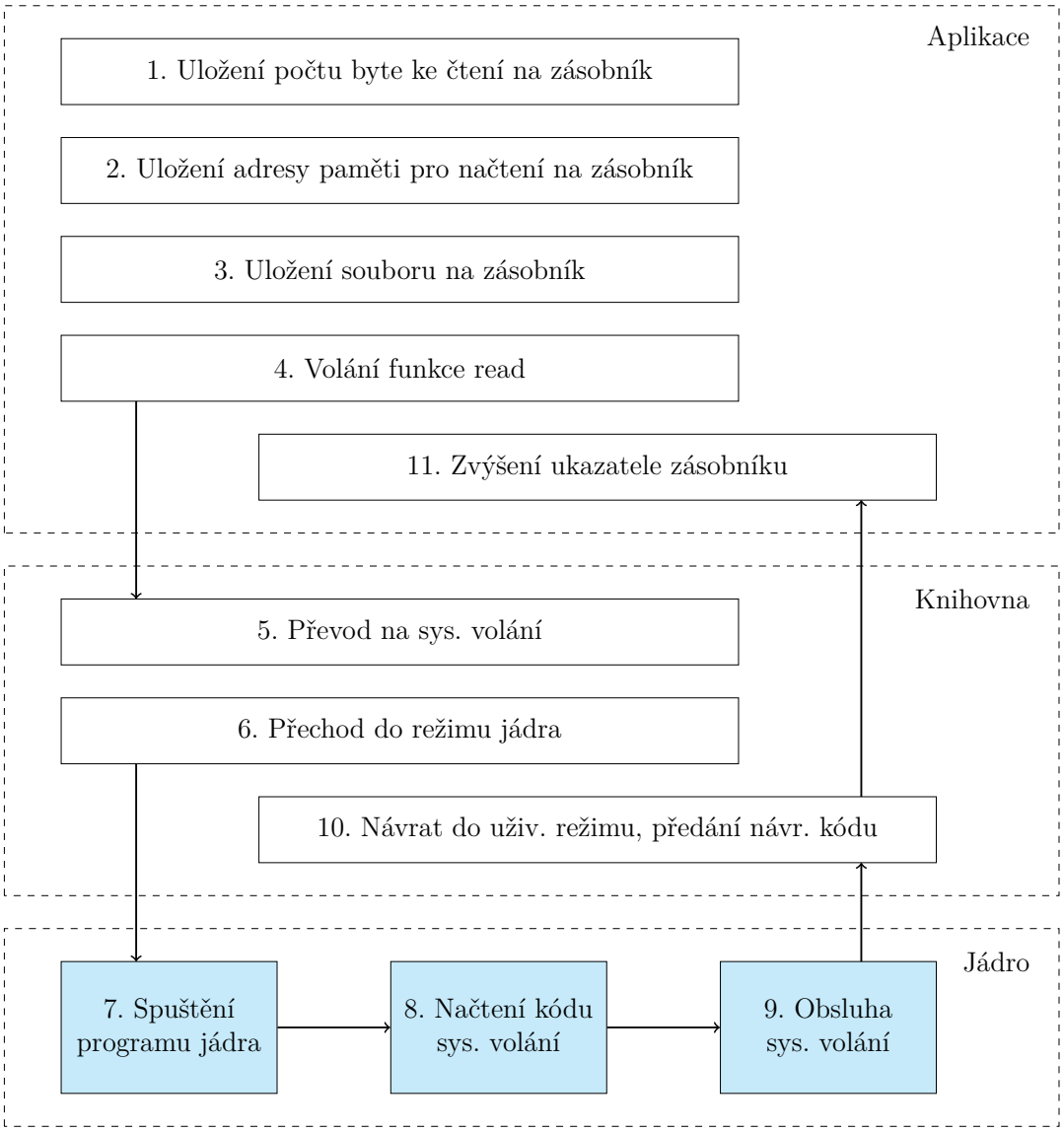
```
1 #include <unistd.h>
2 ssize_t read(int fd, void *buf, size_t count);
```

Výpis kódu 3.7: Funkce knihovny pro čtení dat ze souboru

Pro porovnání toho, co funkce provádí při volání systémového volání jádra pro čtení dat, srovnejte následující popis s výpisem 2.4, který dané akce popisuje v jazyce symbolických instrukcí.

Před voláním knihovní funkce *read* program nejdříve uloží parametry na zásobník[3]. 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.

[3]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 kód systémového volání a jeho parametry (krok 5). Následně knihovna vyvolá softwarové přerušení (krok 6) a spustí tak program jádra (krok 7). Název instrukce pro přechod do režimu jádra se pro jednotlivé architektury liší. Například pro architekturu x86_64 je to *syscall* a pro architekturu ARM to je *swi*.

Program jádra 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 3.8. 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.

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

Výpis kódu 3.8: 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 mít více procesů. Komunikace dvou počítačů v síti je zprostředkována pomocí komunikace mezi procesy, které běží na těchto počítačích.

Operační systém může označit proces **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 výsadní 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]
```

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.

Historicky 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.

Procesy můžeme rozdělit na **systémové** a **uživatelské**. Systémové procesy jsou vytvořeny systémovými programy a jádrem. Uživatelské procesy jsou vytvořeny uživatelskými programy. Příklad rozlišení procesů je uveden ve výpisu 4.3. První tři procesy jsou procesy jádra nebo systémové procesy. Ty se odlišují tím, že pro tyto procesy nejsou známy

[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.

parametry s jakými byly spuštěny a jsou uvozeny hranatými závorkami (není to však exkluzivní podmínka, i jiné procesy mohou být spuštěny tímto způsobem). Další tři procesy jsou uživatelské.

```
[]$ ps xa
968  [xfs-buf/sdb2]
5404 [usb-storage]
96   [cryptd]
1172 /usr/bin/python3
1349 /usr/sbin/NetworkManager
3338 /opt/google/chrome/chrome
```

Výpis kódu 4.3: Systémové procesy a procesy uživatelských programů.

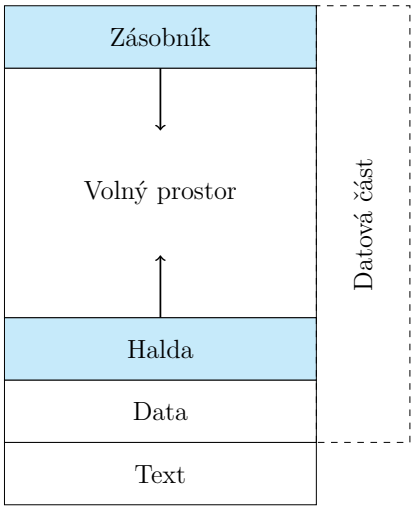
Informace o procesech jsou uloženy v jejich kontextech. V **uživatelském kontextu** jsou informace, které proces potřebuje pro vlastní činnost, například jeho programový kód. V **kontextu jádra** jsou udržovány informace, které potřebuje operační systém, aby mohl proces spravovat. Jedná se například 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 průběh vykonávání procesu. Tato informace je použita pro pozdější obnovení činnosti procesu.

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.



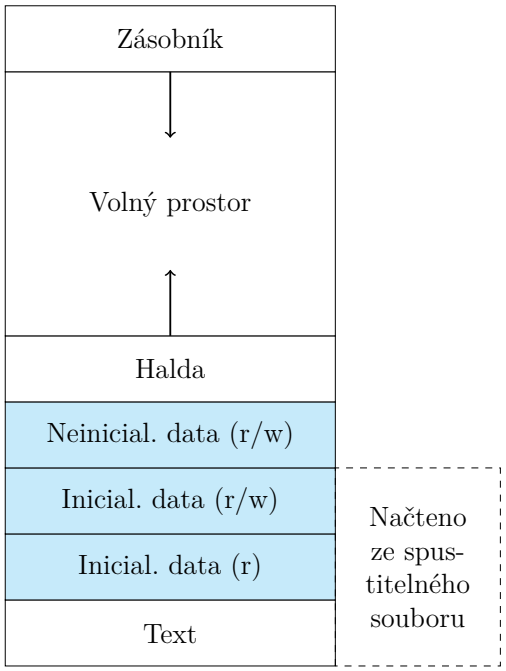
Obrázek 4.1: Uživatelský kontext procesu – statické a dynamické části.

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 – inicializovaná programem, které nelze měnit.
- Inicializovaná data dostupná pro čtení i zápis – inicializovaná programem, lze měnit.
- Neinicializovaná data[3] – lze měnit.

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

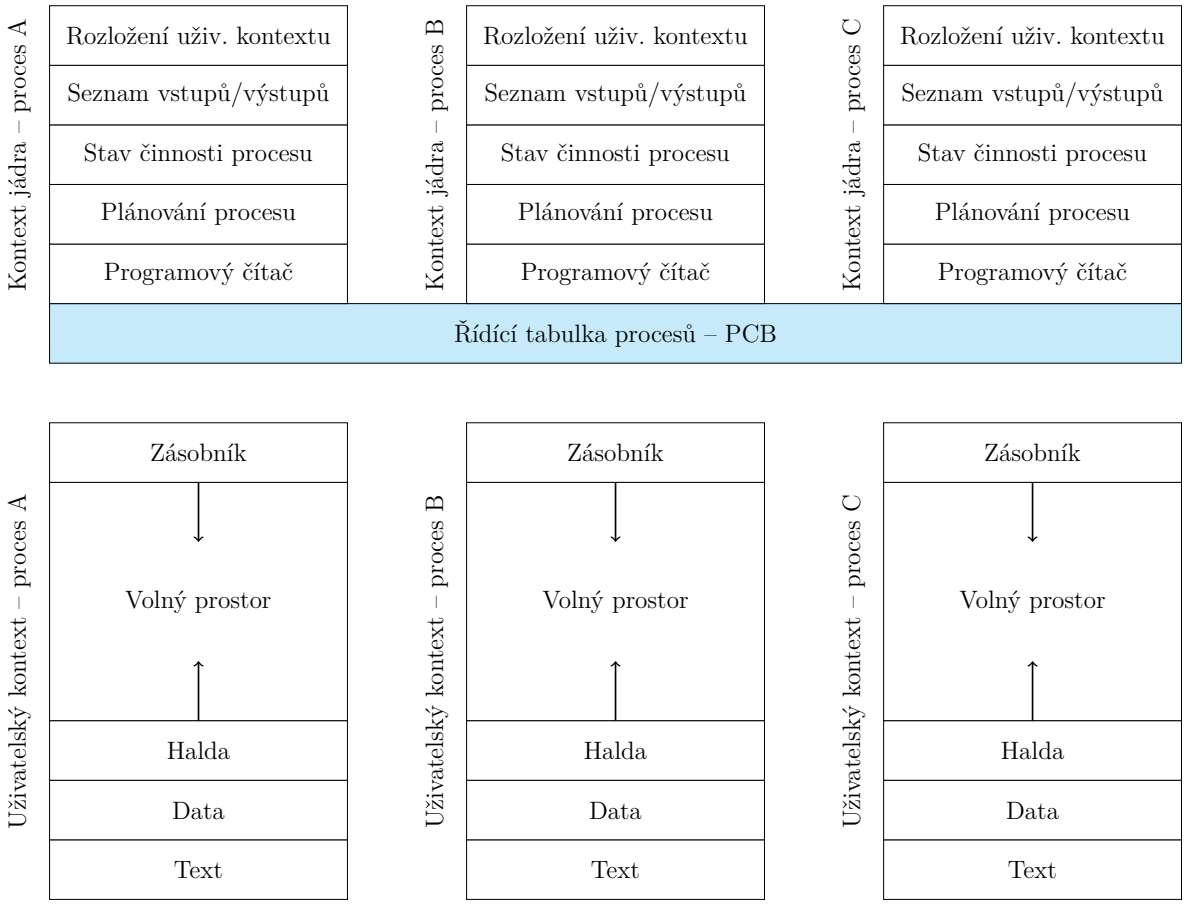


Obrázek 4.2: Datová část paměti procesu.

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“.

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. 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. Například připravený, čekající, a vykonávaný.

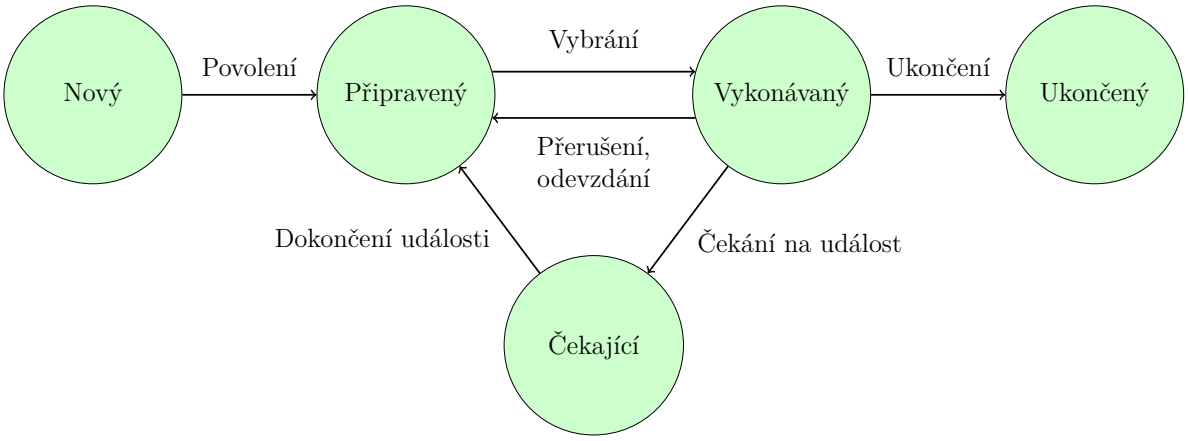
Poslední uvedená informace obsahuje záznam o stavu běhu procesu. Proces se může nacházet v jednom z několika možných základních stavů:

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 kontextu jádra. Dále se načte poslední stav běhu nového procesu z jeho kontextu jádra.

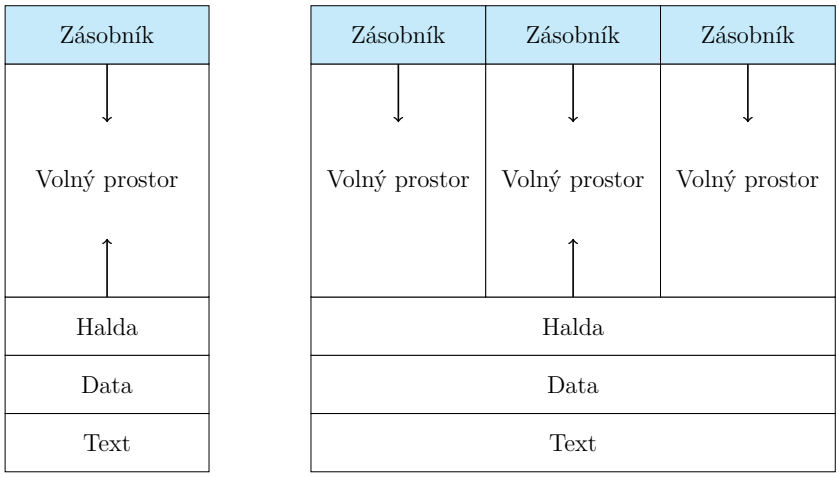


Obrázek 4.4: Základní stavy běhu procesu.

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 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.

[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.



Obrázek 4.5: Uživatelský kontext procesu – jednovláknový a vícevláknový.

Příkladem použití vláken je webový server. Server obsluhuje požadavky velkého počtu klientů. Realizace 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 realizaci s vlákny je pro obsloužení příchozího požadavku spuštěno nové vlákno stejného procesu [16].

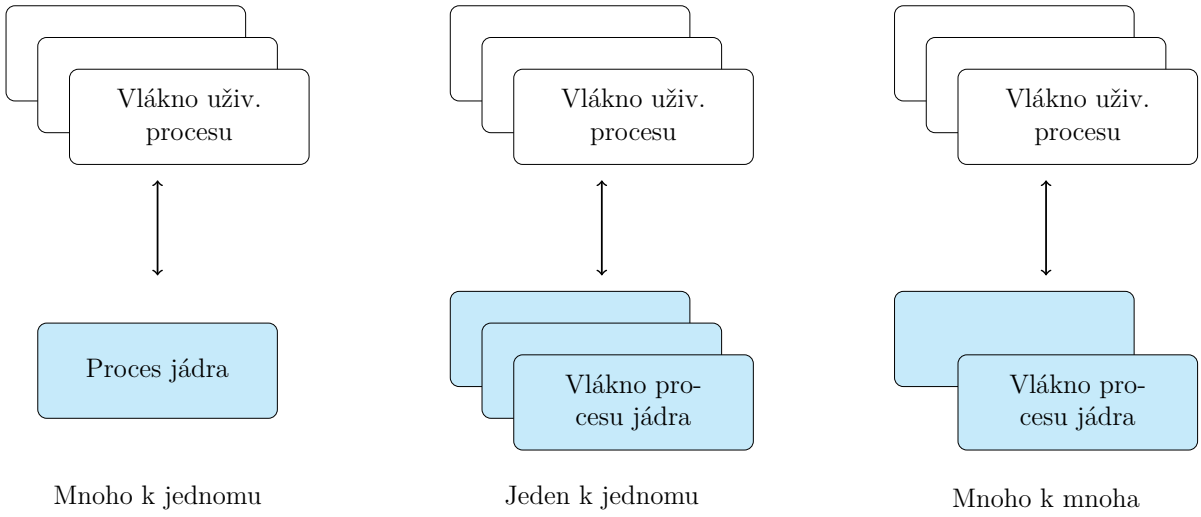
Výhody vícevláknového přístupu lze shrnout následovně:

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

Rozlišují se 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.

[7]Jádro může být také realizováno pomocí vláken.



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. Správa vláken je zajištěna pomocí **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.
- 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. Maximální počet vláken v jádře může být omezen.

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

[8]Neplatí pro všechny operační systémy.

rodičem jednou, ale výsledek volání je vrácen dvakrát: do procesu rodiče a do procesu potomka. Volání vrátí do rodiče PID nového procesu a do potomka hodnotu $PID = 0$. Dalším systémovým voláním (*execve*) se spustí programový kód potomka.

Procesy jsou svázány pomocí identifikátoru PPID (Parent PID), kterým potomek identifikuje rodiče. Ve výpisu 4.4[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 4.4: 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 rodiče 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.

Příklad změny rodiče procesu je zobrazen ve výpisu 4.5. Výpis zobrazuje spuštění skriptu `hierarchy.sh`, který pouze provede příkaz `ping 127.0.0.1`, tedy proces spustí další proces. Výpis zobrazuje, že proces *ping* je potomkem `hierarchy.sh`.

```
[]$ pstree -a | grep -B2 ping
  |   |-bash
  |   |   `-sh hierarchy.sh
  |   |       `-ping 127.0.0.1
```

Výpis kódu 4.5: Hierarchie procesů.

Při ukončení skriptu `hierarchy.sh` je rodič procesu *ping* změněn na hlavní proces *systemd* s $PID = 1$, jak je zobrazeno ve výpisu 4.6. Naopak, pokud se ukončí terminál *bash*, ve kterém byl spuštěn skript `hierarchy.sh` (zde se tiskne výstup příkazu `ping`), tak je ukončen i proces *ping*.

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

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

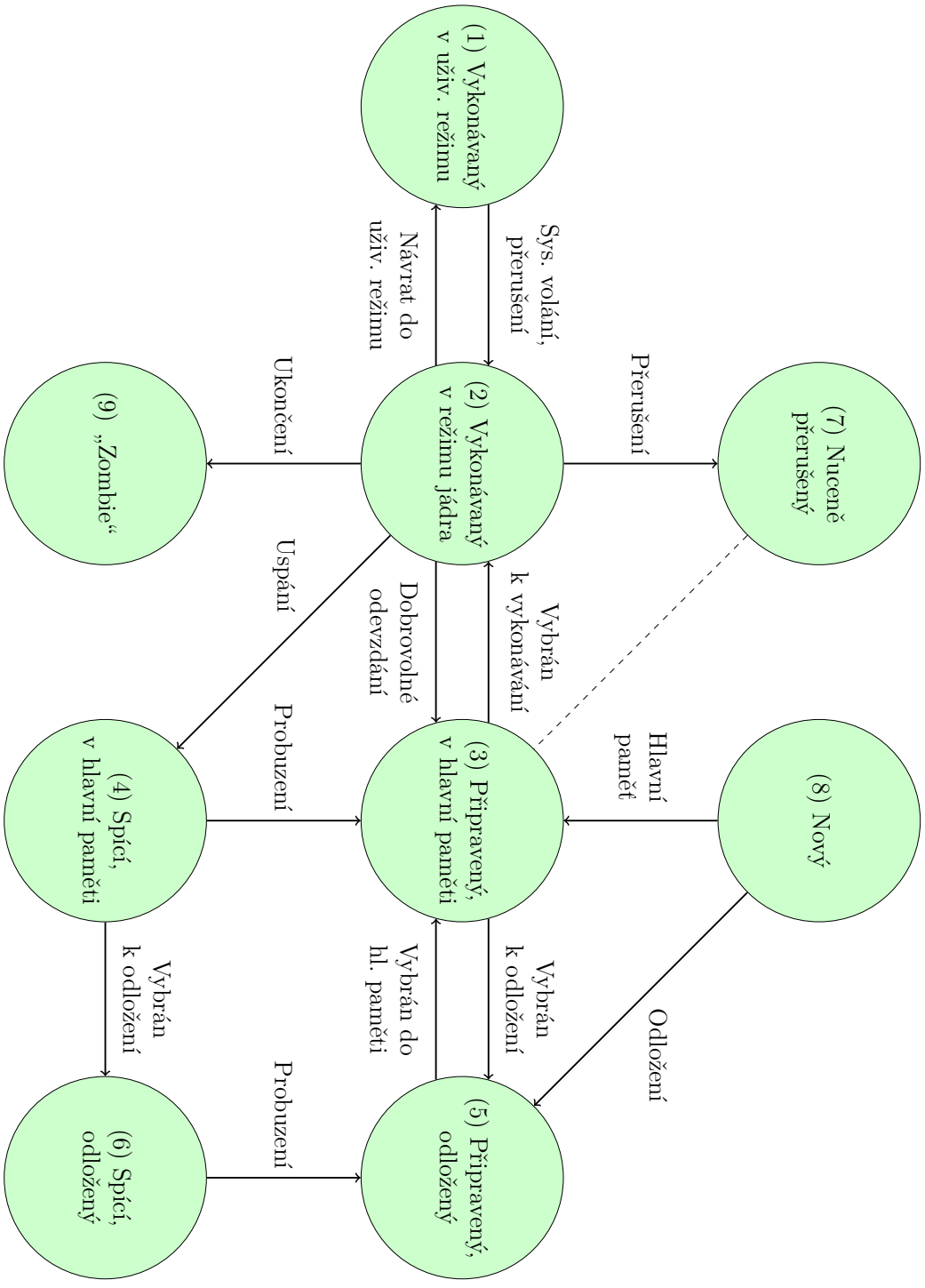
Výpis kódu 4.6: Zdědění procesu.

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 v uživatelském režimu
2. Proces je vykonáván v režimu jádra
3. Proces je připraven ke zpracování a je v hlavní paměti
4. Proces je spící a je v hlavní paměti
5. Proces je připraven ke zpracování a je v odkládací paměti
6. Proces je spící a je 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

[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.

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.
- 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.

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

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 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

[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.

(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.

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. Historicky bylo úkolem rozdělit několik procesů na jeden procesor. Nyní plánování pracuje s rozdělováním více procesů na více procesorů.

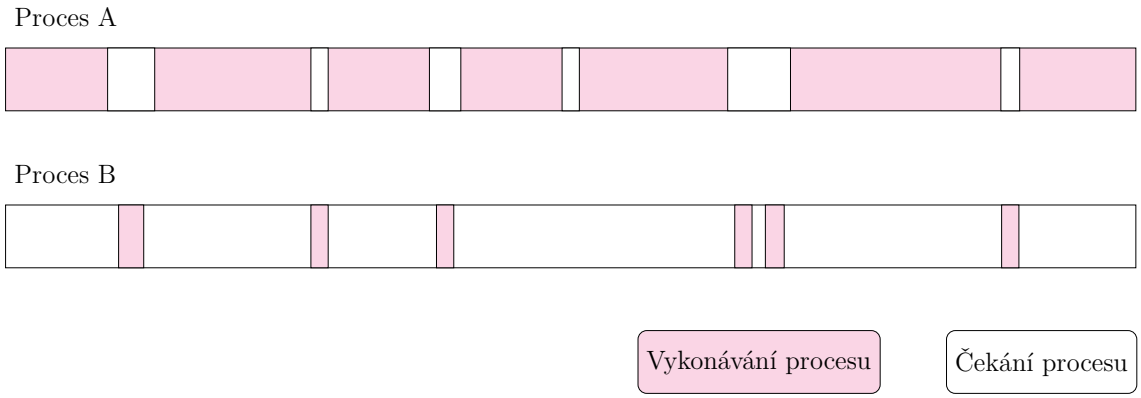
Co však dělá situaci tak komplikovanou, že nelze použít jednoduché „spravedlivé“ střídání procesů? I tato možnost by fungovala. Představme si však situaci, že používáme interaktivní systém a v něm existují dvě soupeřící úlohy. První aktualizuje obrazovku po uzavření okna a druhá odesílá email. Pokud 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 dojem uživatele ze systému bude dobrý.

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 – bude vykonáván proces rodiče nebo potomka?
2. Ukončení procesu – bude vybrán jiný proces k vykonávání.
3. Blokování procesu – bude vybrán jiný proces k vykonávání.
4. Přerušení od zařízení – bude vykonáván proces, který čekal na data ze zařízení, nebo pokračovat jiný proces?
5. Nucené nebo dobrovolné přerušení procesu – bude vybrán jiný proces k vykonávání.

První dva okamžiky rozhodnutí jsou svázány s vytvořením a ukončením procesu a jsou jednorázové. Okamžik rozhodnutí při blokování procesu může nastávat často podle typu procesu. 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 typů procesů je vidět na obrázku 4.8. První typ procesu je více výpočetně zaměřený, druhý provádí hodně I/O operací. Růst rychlosti CPU rychlejší než rychlost paměťových médií. Procesy tedy většinově čekají (na obrázku proces B).

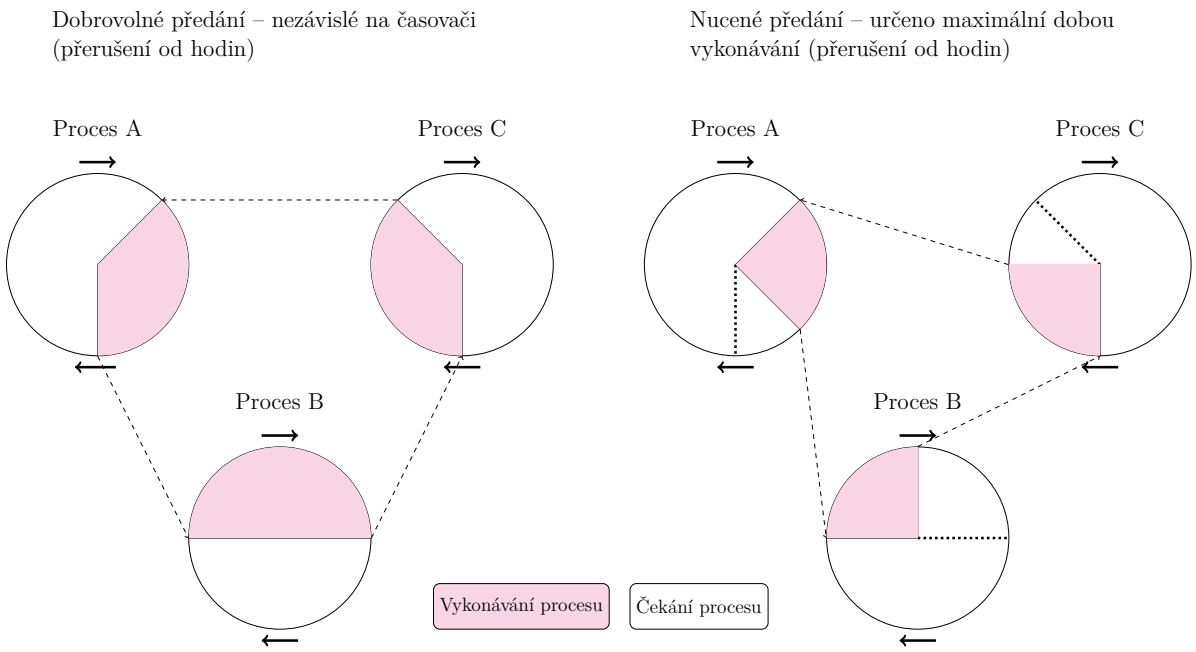


Obrázek 4.8: Střídání výpočetního intervalu s intervalem čekání pro a) proces s dlouhými intervaly výpočtů b) proces s převažujícími I/O operacemi.

Konec I/O operace je signalizován pomocí přerušení od zařízení. Při tomto okamžiku rozhodnutí musí plánovač zvolit, zda bude vykonáván proces, který byl blokovaný a čekal na data ze zařízení, nebo bude pokračovat vykonávání aktuálního procesu.

Rozlišujeme nucené a dobrovolné přerušení procesu. Proces je nuceně přerušen poté, co byl vykonávaný po maximální dobu, která je signalizována přerušením od hodin. Nastane okamžik rozhodnutí, kdy je nutno vybrat jiný proces k vykonávání. Tato situace nastává v **preemptivních** [13] systémech. K dobrovolnému přerušení dochází, když proces sám pozastaví své vykonávání. Po dobrovolném přerušení nastane okamžik rozhodnutí a je vybrán jiný proces k vykonávání. Tato situace nastává v **kooperativních** systémech. Rozdíl mezi preemtivním a kooperativním systémem je zobrazen na obrázku 4.9.

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



Obrázek 4.9: Kooperativní a preemptivní přepínání procesů.

4.5.2 Typy systémů

Počítačové systémy slouží různým účelům 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. Například otevření nebo uzavření okna by mělo být provedeno prioritně před stahováním dat ze sítě. Interaktivní operační systémy se používají u osobních počítačů. Plánovač nemá vliv na složitost úlohy a rychlost jejího zpracování. Může však splnit 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 uživatelem do obdržení výsledku.
- Přiměřenost – vztah složitosti úlohy k době jejího zpracování.

Uživatelé mají 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 jednoduchou úlohou (vnímanou jako jednoduchou). 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 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 je použita statická alokace paměti. U dynamické alokace se udržují seznamy volných bloků paměti, jejichž prohledání a zabrání přináší časové zdržení. 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 z důvodu chyby v aplikaci. Tato situace není při statické alokaci možná. Únik paměti (memory leak) může po delší době vést ke kolapsu systému.

U real-time systémů jsou sledovány tyto parametry:

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

Například v 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 výstupu (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]:

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

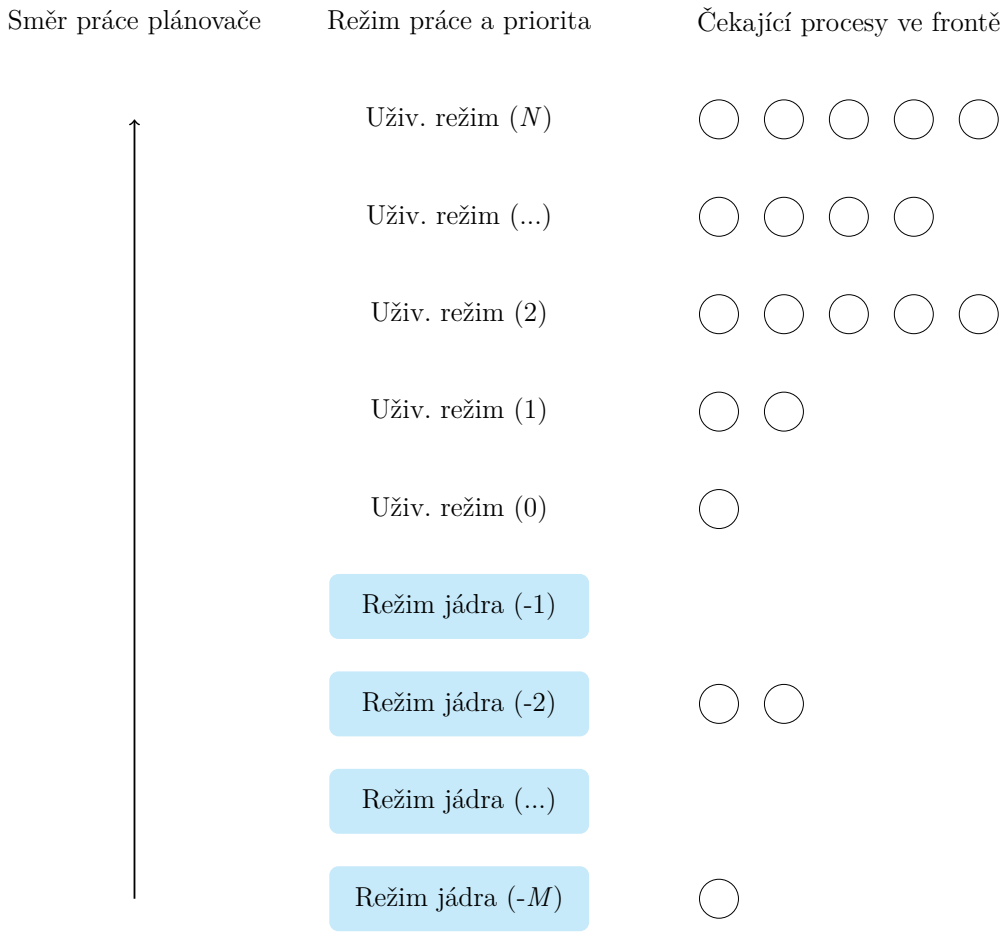
Jako příklad uvažme situaci se třemi periodickými událostmi $m = 3$ s těmito periodami výskytu $P_1 = 100$, $P_2 = 200$ a $P_3 = 500$ ms. Tyto události vyžadují pro svou obsluhu $C_1 = 50$, $C_2 = 30$ a $C_3 = 100$ ms. Systém vyhovuje, protože $0,5 + 0,15 + 0,2 < 1$.

4.5.3 Činnost plánování

Obecné plánování procesů pracuje ve dvou úrovních [11, 27]:

- 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 prioritu kladnou. Naopak procesy v režimu jádra mají prioritu zápornou. Negativní hodnoty značí vyšší prioritu a pozitivní hodnoty značí nižší prioritu. Toto 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 procesem, vybere první proces z této fronty. Vybraný proces je vykonáván a poté vložen zpět do své fronty. S procesy v rámci jedné fronty je zacházeno cyklicky (round-robin). Priorita P každého procesu je intervalově přepočítána (například každou sekundu) podle vzorce [11]:

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

Proces je následně přesunut do příslušné fronty podle přepočtené priority.

- Hodnota V představuje využití procesoru. V je zvyšováno o takty hodin, při kterých je proces vykonáván. Zvyšováním hodnoty V se proces posunuje do fronty s nižší prioritou. Tímto způsobem jsou starší (z pohledu vykonávání) 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 hodin 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 tím, že čeká na I/O operaci. Proces je odstraněn ze své plánovací fronty. Až nastane událost, na kterou proces čekal, je vložen do plánovací fronty podle jiné priority. Volba fronty závisí na události, na kterou proces čekal – hodnota B).

Příklad konkrétních hodnot priority pro procesy je zobrazen ve výpisu 4.7. Sloupec *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 4.7: Hodnoty priority procesů.

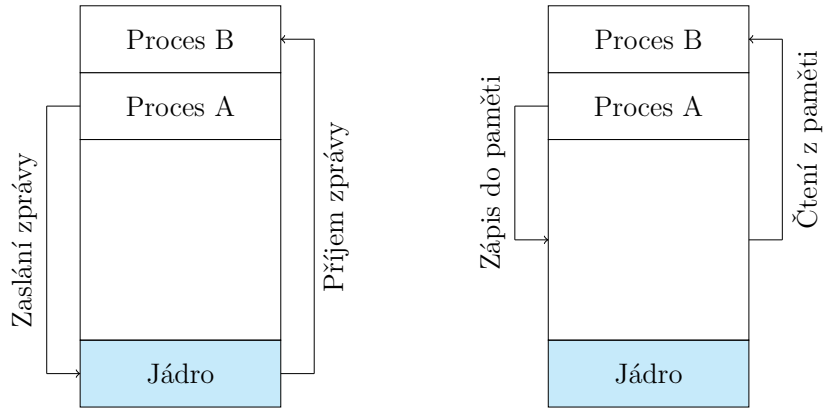
4.6 Komunikace mezi procesy

Procesy mohou pracovat nezávisle nebo mohou spolupracovat. Nezávislý proces nekomunikuje s jinými procesy – nevyměňuje si data a zprávy. Spolupracující proces komunikuje s jinými procesy. Výměna dat mezi procesy je prováděna pomocí **sdílené paměti** a **zasílání zpráv** [11, 16].

- Sdílená paměť – v paměťovém prostoru je vyhrazena sdílená oblast. Procesy si vyměňují informace čtením a zápisem z/do této sdílené paměti.
- Zasílání zpráv – informace jsou vyměňovány pomocí zaslání dat. Zprávy mohou být posílány přes systémová volání.

Oba způsoby jsou uvedeny na obrázku 4.11. U předávání zpráv je zobrazen případ komunikace pomocí systémového volání. V případě sdílené paměti je informace procesem A zapsána na místo ve sdílené paměti a proces B si ji následně přečte.

[14] Překládá se nejčastěji jako „ohleduplnost“.



Obrázek 4.11: Způsoby komunikace mezi procesy pomocí systémového volání a sdílené paměti.

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.

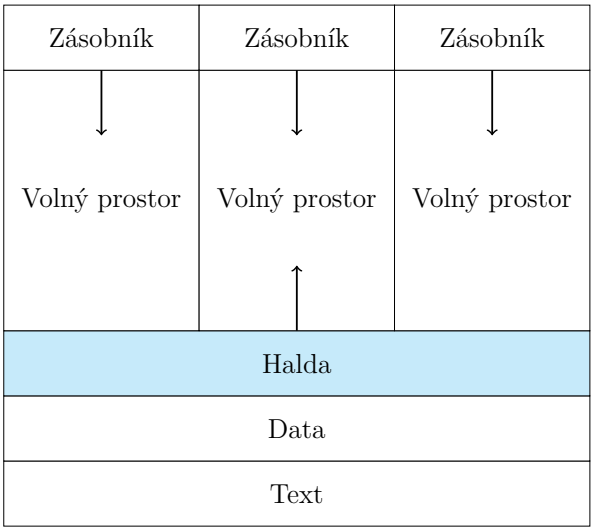
Příkladem zasílání zpráv je propojení výstupu jednoho procesu na vstup druhého procesu pomocí **roury** (pipe). Je zde použit blokující režim – proces, který chce zprávu číst data je blokován, dokud není zpráva k dispozici (dokud nepřijme signál, že je k dispozici). Roura se zapisuje jako `cmd1 | cmd2`,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

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

Rouru lze použít pro výpis adresáře po stránkách pomocí příkazu `ls | more`. Zde je vytvořena dočasná paměť, do které první příkaz zapisuje zprávu. Jak je přijímacímu procesu oznámen konec zprávy – zaslán signál, tak proces začne zprávu číst. Po ukončení práce obou procesů je dočasná paměť uvolněna. Další možností je komunikace pomocí zápisu do souboru, tzv. pojmenovaná roura. Zde 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 pak zapisovat zprávu do roury pomocí přesměrování klávesnice do souboru `cat > roura`. V prvním terminálu bude zpráva zobrazena po ukončení zápisu zprávy do roury (znak nového řádku).

Sdílená paměť

Sdílená paměť je oproti zasílání zpráv vhodná pro přenos velkého objemu dat. Procesy si sdílený blok paměti přičlení do uživatelského kontextu. Pokud do paměti zapíše jeden proces, jsou data ihned viditelná ostatním procesům. Dalším způsobem je sdílená paměť pro vlákna 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.

Sdílená paměť 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 více procesů (vláken), které mezi sebou komunikují. Proces může ovlivnit jiný proces, nebo být ovlivněn. V této kapitole se zaměříme na zajištění synchronizace procesů. Seznámíme se i s klasickým problémem synchronizace.

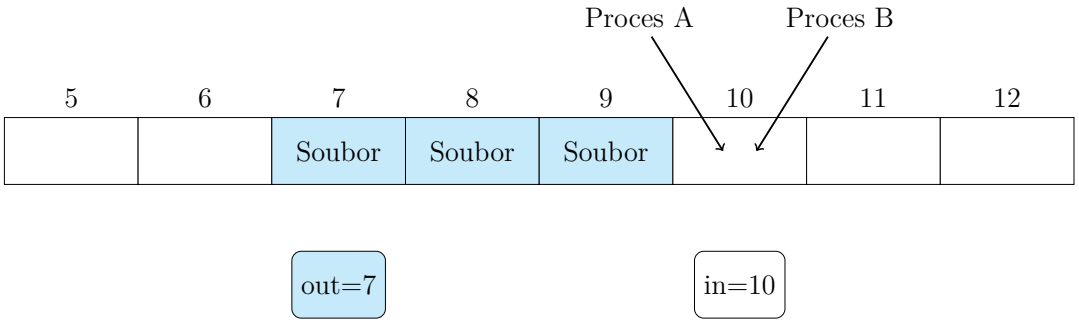
4.7.1 Požadavky synchronizace

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

- vyloučení konfliktu kritických činností,
- zajištění časových mezníků.

Konflikt kritických činností

Proč potřebujeme vyloučit konflikt kritických činností? Důvod je práce procesů se sdílenými prostředky. Představme si situaci tiskové fronty. Proces uloží soubor do 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.

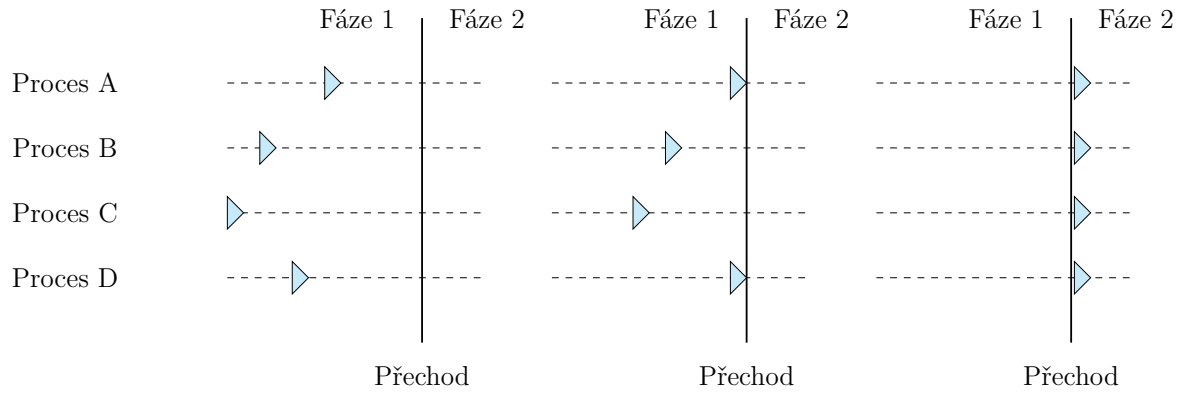
Tiskový démon vytiskl několik souborů a ukazatel *out* je nastaven na 7. Ukazatel *in*, ukazující na první volné místo, má hodnotu 10. Dva procesy se „rozhodnou“ téměř ve stejném okamžiku uložit do fronty svůj soubor pro tisk. Proces A si přečte ukazatel *in* a hodnotu 10 uloží do své lokální proměnné *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 slot 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 slotu č. 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 přístupu, se nazývá **souběh** (race condition). Pokud 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 zápis. Dle Murphyho zákonů program v době 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 souběhu.

[15]Nebo při obhajobě BP/DP.

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 vláken procesu. Používají se **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 velkých obrazových dat, kdy vlákna jednoho procesu zpracovávají části obrazu. Na konci každé fáze musí být vytvořena podoba celého obrazu pro nadcházející fázi.



Obrázek 4.14: Časová souslednost procesů.

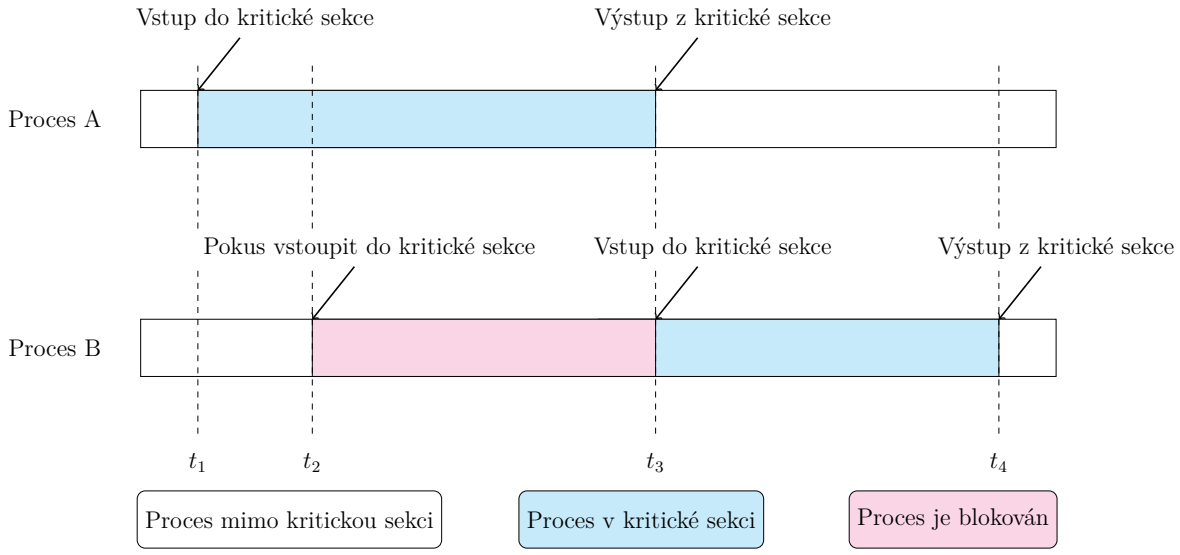
4.7.2 Vzájemné vyloučení a kritická sekce

Dále se budeme zabývat prvním a náročnějším požadavkem synchronizace – zamezením souběhu. Připomeňme si, že souběh je situace, kdy dva nebo více procesů přistupují ke sdílenému prostředku a výsledek závisí na tom, v jakém pořadí a čase k nim přistoupí.

Jak lze zabránit souběhu? Pomocí vyloučení čtení a zápisu sdílených dat ve stejném okamžiku – tzv. **vzájemné vyloučení** (mutual exclusion). Pouze jeden proces může přistupovat ke sdílenému prostředku po dobu jeho práce s tímto prostředkem. Pokud jiný proces žádá o využití stejného prostředku, přejde do stavu čekání dokud prostředek není uvolněn

Vzájemné vyloučení řešíme tak, že ve zdrojovém kódu programu je vymezena část kódu, která přistupuje ke sdílenému prostředku. 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.

Příklad dvou procesů s řízeným přístupem do 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 selže, neboť v kritické sekci je již jiný proces (proces A). Proces B je 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 je do zdrojového kódu zařazena vstupní a výstupní sekce, viz výpis 4.8:

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

Výpis kódu 4.8: 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. Dále 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 procesem, který pracuje mimo svou kritickou sekci.

2. Doba čekání na vstup do kritické sekce musí být konečná.

4.7.3 Vzájemné vyloučení pomocí proměnných

Vzájemné vyloučení pomocí proměnných je realizováno pomocí sdílené dvouhodnotové proměnné a striktního střídání procesů [11, 16].

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 4.9.

```
1  -- Proces A --
2  while (TRUE) {
3   while (turn != 0) {}  /*cekej az na mne bude rada*/
4   kriticka sekce
5   turn = 1;
6   ne kriticka sekce
7  }
8  -- Proces B --
9  while (TRUE) {
10  while (turn != 1) {}  /*cekej az na mne bude rada*/
11  kriticka sekce
```

```
12  turn = 0;
13  ne kriticka sekce
14 }
```

Výpis kódu 4.9: 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 jeho přerušení. Proces může být přerušen i z jiného důvodu, např. při obsluze vstupně/výstupního zařízení. Řešením vzájemného vyloučení je zakázání všech přerušení, pokud je nějaký proces ve své kritické sekci. Pokud není přerušení povoleno, tak vykonávaný proces nemůže být přerušen a jiný proces nemůže vstoupit do své kritické sekce. Přerušení mohou teoreticky zakazovat uživatelské a systémové programy. Umožnění uživatelským procesům zakazovat přerušení není moudrý přístup, neboť tyto procesy mohou způsobit pád systému. Uživatelské programy přidávají uživatelé a je jejich volbou, který program instalují. Pokud není program dobře napsán (nebo zlomyslně chybu obsahuje), může zablokovat všechna přerušení a pak je nikdy neobnovit. Možností je zákaz přerušení pouze pro systémové procesy.

Zakázání přerušení pracuje jen u jednoprocesorových systémů. Pokud je na jednom procesoru zakázáno přerušení, tak tento zákaz není platný pro jiný proces vykonávaný na jiném procesoru. Při zákazu 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

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

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

Uvažme dva procesy s rozdílnými prioritami, proces H s vyšší prioritou a proces L s nižší prioritou. Plánování procesů 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 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 zařídit vzájemné vyloučení 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 velikost pro N položek. Producent položky 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 4.10 a proces spotřebitele je zobrazen ve výpisu 4.11. Proměnná *pocet* obsahuje počet položek v paměti. 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;
```

```
14   if (pocet == 1) wakeup(spotrebitel); /*pamet byla prazdna -> probud
      spotrebitele*/
15  }
16 }
```

Výpis kódu 4.10: Producent.

```
1  void zakaznik(void)
2  {
3   int polozka;
4   while (TRUE)
5   {
6    if (pocet == 0) sleep() /*bez spat, nic neni v pameti*/
7    polozka = odeber_polozku(); /*vyjmi polozku z pameti*/
8    pocet = pocet - 1;
9    if (pocet == N - 1) wakeup(producent) /*pamet byla plna -> probud
      producenta*/
10   spotrebuj_polozku(polozka);
11  }
12 }
```

Výpis kódu 4.11: Spotřebitel.

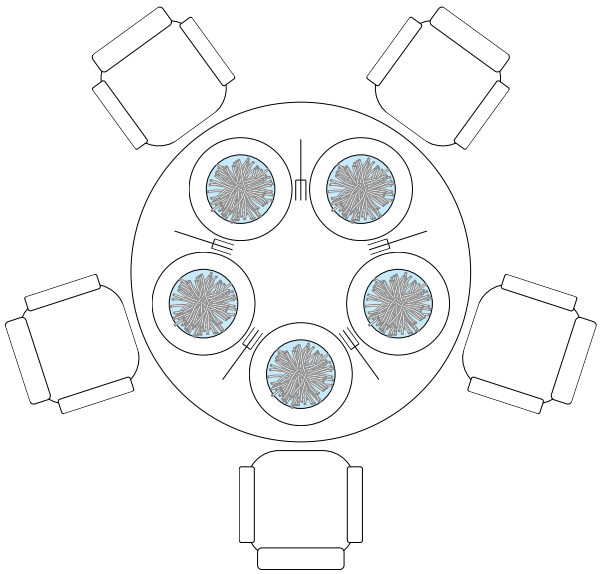
Problém tohoto řešení je **fatální souběh** (fatal race condition). Předpokládejme, že paměť je prázdná a spotřebitel přečetl hodnotu proměnné $pocet = 0$. V tomto okamžiku bylo přerušeno vykonávání procesu spotřebitele a začal se vykonávat proces producenta. Proces spotřebitele se neuspal. Producent vloží položku do paměti a zvýší jejich počet o jedna na $pocet = 1$. Producent předpokládá, že spotřebitel byl ve stavu spícím. Proto zavolá systémové volání *wakeup*, aby spotřebitele probudil. Avšak spotřebitel nebyl ve stavu spícím, a tato zpráva o probuzení je ignorována (ztracena). Časem se začne vykonávat proces spotřebitele. Spotřebitel má již z minulého vykonávání přečteno $pocet = 0$ a další krok je, že se uspí. Opět se začne vykonávat proces producenta, ten zaplní paměť položkami a uspí se. Oba procesy se tak dostanou do stavu spánku a spí jako Šípková Růženka; jsou ve fatálním souběhu.

Konečným řešením vzájemného vyloučení jsou semafory a monitory, viz příklad 4.9.4.

- Semafory poskytuje operační systém (realizováno jádrem). Jsou tedy nezávislé na programovacím jazyce.
- Monitory poskytují knihovny programovacího jazyka. Jsou to abstraktní datové typy, které zahrnují 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 vznikly zajímavé problémy. Jedním z nejznámějších je problém večeřících filozofů „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. Stůl je zobrazen na obrázku 4.16.



Obrázek 4.16: Stůl s večeří pro pět filosofů.

Filosofové při 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 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 byl odvozen od problému hladovějících filosofů. Výskyt hladovění lze redukovat volením náhodných časů čekání mezi jednotlivými pokusy. Toto řešení je efektivní a v praxi funguje tam, kde odložení požadovaného úkonu není problém. Příkladem z oblasti síťové komunikace je přenos technologií Ethernet – při kolizi přenosu stanice čekají náhodnou dobu, aby pokus zopakovaly. Ovšem ve 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 pořadí určeno není. 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

Programy mohou navodit stav, kdy 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 jsou dva procesy, kdy každý chce vypálit na CD dokument uložený ve skeneru. První proces zažádá o skener a ten 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.Prostředky dělíme na **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 je uzamčení sdílitelného prostředku pro zajištění vzájemného vyloučení.

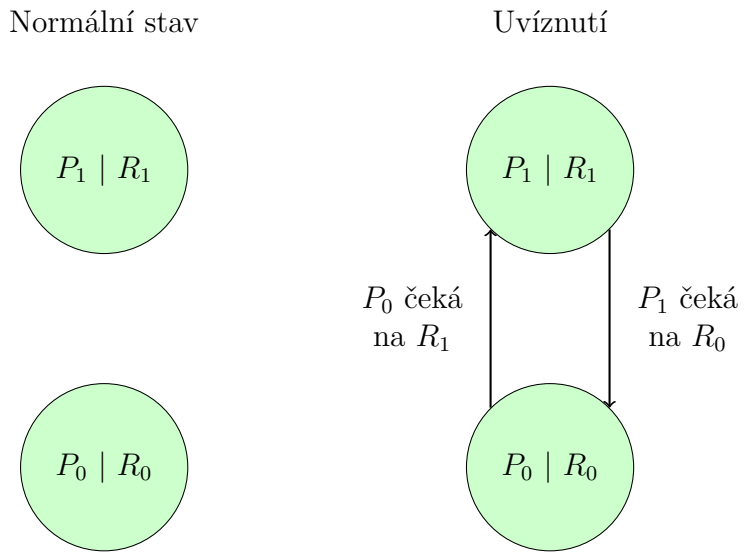
Prostředky jsou také **přepínatelné** (preemptable) a **nepřepínatelné** (nonpreemptable). Odebrání přepínatelného prostředku procesu nezpůsobí škodu. Příkladem tohoto prostředku je hlavní paměť. Přidělená paměť může být procesu odebrána a data procesu přesunuta do odkládací paměti (viz princip virtuální paměti). Odebrání nepřepínatelného prostředku procesu způsobí škodu. Pokud proces začne vypalovat CD, odebrání CD-RW mechaniky způsobí poškození dat.

Uvíznutí nenastává nad sdílitelnými prostředky bez jejich uzamčení a nad přepínatelnými prostředky, kde lze uvíznutí předejít předáním prostředku jinému procesu. Při popisu uvíznutí budeme uvažovat prostředky nesdílitelné, tj. může je využívat pouze jeden proces v jednom čase, a nepřepínatelné, tj. jejich odebrání procesu způsobí škodu. Využívání těchto prostředků se skládá ze tří kroků:

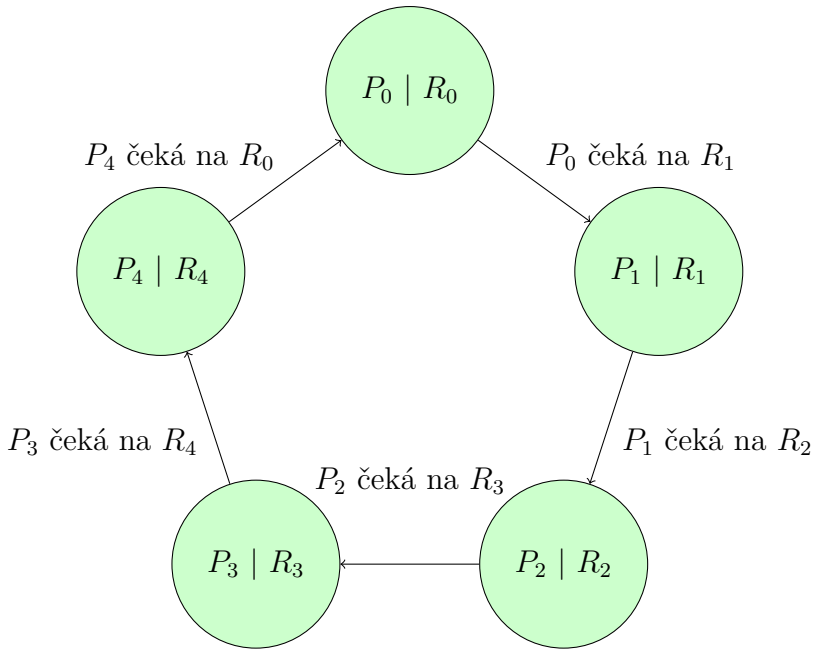
1. Žádost – Proces požádá o prostředek. Pokud je prostředek využíván, přejde proces do stavu čekání na uvolnění prostředku.
2. Používání – Proces využívá prostředek.
3. Uvolnění – Proces po ukončení práce předá zprávu o uvolnění prostředku.

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. Tento stav může vzniknout u dvou procesů, viz obrázek 4.17,

nebo u skupiny procesů, viz obrázek 4.18. K uvíznutí může také dojít při nekorektním chování (chyba v programu), například při ztrátě předávané informace o uvolnění prostředku.



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



Obrázek 4.18: Uvíznutí skupiny procesů P nad prostředky R.

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

1. předcházení uvíznutí,

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

Předcházení uvíznutí může být 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.

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 jsou ukončeny. Ukončení procesů 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 ovšem se ztrátou dat.
2. Procesům jsou odebírány prostředky a předány 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, včetně stavu „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 rozhraní pro přístup k informacím o činnosti jádra. Každý proces svůj podadresář odpovídající jeho PID. V podadresáři je několik souborů, jejichž obsah zobrazuje infomrace o činnosti procesu. Obsah souborů je dynamicky generován na základě požadavku čtení. Například v souboru `cmdline` je název

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

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 4.12. 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 4.12: Čas vytvoření procesu.

Stav procesu je dostupný v souboru `status`, jehož obsah je zobrazen ve výpisu 4.13. Zde je také uveden počet vláken procesu.

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

Výpis kódu 4.13: Stav jednoho procesu.

Přehled stavů více procesů lze získat příkazem `ps`, viz výpis 4.14. 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 4.14: 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 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 4.15, 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");
```

```
12              printf("PID %d\n", getpid());
13              exit(0);
14      /*rodic - vraceno PID potomka*/
15      default: sleep(60);
16              exit(0);
17      }
18  }
```

Výpis kódu 4.15: Vzorový program pro uživatelský kontext a stav zombie.

Program po kompilaci (`gcc zombice.c -o zombice`) má uživatelský kontext zobrazený ve výpisu 4.16. 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 4.16: 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 oddíly (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 4.17. 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 4.17: 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 4.18. 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 4.18: Textové řetězce ve zdrojovém kódu programu.

Nyní prozkoumejme obsah části *.rodata* s textovými řetězci, jak je uvedeno ve výpisu 4.19. Na začátku výpisu je přidána informace o architektuře, pro kterou byl program zkompilován. Následuje zobrazení textových řetězců uvedených ve zkompilovaném programu `zombice`. Program `objdump` může také být využit pro zobrazení zdrojového kódu v jazyce symbolických instrukcí. Tyto a další informace mohou být využity pro **reverzní inženýrství** (reverse engineering), kdy je potřeba „vytáhnout“ co nejvíce informací ze zkompilovaného programu. Například se dá zjistit, jakými vstupními parametry lze program spustit (pokud nejsou známy), či zda tam jsou nějaké parametry 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 4.19: Zobrazení sekce *.rodata* (textové řetězce) přeloženého binárního souboru.

Připomeňme si, že data uživatelského kontextu dělíme 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“).

Část neinicializovaných dat *bss*, nebyla dosud ve zdrojovém kódu programu `zombice` zastoupena. Ve výpisu 4.16 je uvedena minimální velikost sekce (4 B). Jako další experiment provedeme to, že do zdrojového kódu přidáme neinicializovaná data (datové pole), tak jak je zobrazeno ve výpisu 4.20. 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));
```

```
7
8   /*vytvoreni noveho procesu*/
9   switch (fork())
10   ...
```

Výpis kódu 4.20: 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 4.21.

```
[]$ 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 4.21: 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 4.16 (hodnota nárůstu je přibližná z důvodu práce s bloky paměti).

Nyní prozkoumejme celkovou velikost souboru s programem (ne jenom uživatelského kontextu), viz výpis 4.22. 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 4.22: Porovnání velikosti souborů s přeloženým programem – původní a upravený.

Program `zombice` zužitkujme pro navození stavu „zombie“, což je obsahem vlastního zdrojového kódu. Stav vznikne tak, že proces rodiče vytvoří proces potomka pomocí funkce *fork*. Proces rodiče 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 rodiče má tedy stále informaci, že proces potomka 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 a ten přijme zprávu o ukončení potomka. Stav „zombie“ pak 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 zpráv probíhá v blokujícím nebo neblokujícím režimu. Blokující režim značí, že vysílací proces čeká, dokud není zpráva přijata; přijímací proces čeká, dokud není zpráva dostupná. Zprávy v blokujícím režimu lze přenášet pomocí roury. Proces do roury zapíše blok dat. Následuje značka (znak nového řádku), že přijímací proces může data číst. Roury mohou také 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 ukončení zápisu 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 problému práce s pamětí[17], nebo je může zasílat uživatel.

Uživatel může signály zasílat pomocí zadání znaků do terminálu. Signál SIGTSTP zaslaný pomocí [Ctrl+z] pozastaví proces na popředí terminálu a převede jej na pozadí terminálu. Seznam pozastavených procesů lze získat pomocí příkazu `jobs`. Pozastavený proces lze obnovit a přenést na popředí terminálu, 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 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 (proces) ukončí.

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

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

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

```
9
10 #nekonecna smycka
11 while true
12 do
13  sleep 1
14 done
```

Výpis kódu 4.28: Zachycení signálu.

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; neukončí potomky procesu.
- SIGKILL – násilně ukončí proces; signál nemůže být zachycen; určeno pro ukončení procesu v případě problému; ukončí potomky procesu.

Příklad použití signálu SIGTERM[18] pro nenásilné ukončení procesu a signálu SIGKILL pro násilné ukončení procesu je uveden ve výpisu 4.29.

```
1  #nenasilne ukonceni - lze zachytit - signal SIGTERM
2  []$ sh signal.sh &
3  []$ ps a
4  PID  TTY      STAT   TIME COMMAND
5  9374 pts/0    S      0:00 sh signal.sh
6  []$ kill 9374
7  #SIGTERM je vychozi signal prikazu kill
8  []$ ps a
9  ...
10 [1]+  Ukoncen (SIGTERM) sh signal.sh
11
12 #nasilne ukonceni - nelze zachytit - signal SIGKILL
13 []$ sh signal.sh &
14 []$ ps a
15 PID  TTY      STAT   TIME COMMAND
16 9245 pts/0    S      0:00 sh signal.sh
17 []$ kill -9 9245
18 #cislo 9 = signal SIGKILL
19 []$ ps a
20 ...
21 [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.

[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

```
[]$ 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.

4.9.4 Souběh procesů

Příklad demonstruje synchronizaci procesů při práci se sdílenou proměnnou. Situace, kdy dva nebo více procesů přistupují ke sdílenému prostředku a výsledek závisí na pořadí a času přístupu se nazývá souběh. Souběhu lze zabránit pomocí vyloučení čtení a zápisu sdílených dat ve stejném okamžiku. Ve zdrojové kódu programu je část, která ovládá přístup ke sdílenému prostředku (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, kterým je proměnná typu „interger“. Hodnota 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 zobrazením 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.

```
import multiprocessing, os

def odeber(sprom):
    print('Proces A ID:', os.getpid())
    #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
    for i in range(1000): sprom.value -= 1

def pridej(sprom):
    print('Proces B ID:', os.getpid())
    #vytvoreni delsiho casu behu, aby mohlo dojit k preruseni
    for i in range(1000): sprom.value += 1

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

#vytvorim procesy
```

regulární výraz. Tečka v regulárním výrazu značí shodu s jakýmkoliv znakem; je potřeba ji vyčlenit pomocí znaku `\.`. Dále jsme vynechali část s mezerou. Ta je v regulárním výrazu značena jako `[[:space:]]`.

```
A = multiprocessing.Process(target=odeber, args=(sprom,))
B = multiprocessing.Process(target=pridej, args=(sprom,))

#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 4.31: Práce se sdílenou proměnnou.

Souběh lze řešit několika způsoby. Jedním z nich je použití semaforů, které jsou nezávislé na programovacím jazyce. Semafory poskytuje operační systém a jsou založeny na předávání zpráv přes jádro systému. Výpis 4.32 zobrazuje použití semaforu pro opravu předchozího programu. V kódu je vytvořen semafor *zamek*. Tento semafor je předaný jako parametr při vytváření procesů A a B. Před vstupem do kritické sekce procesů A a B je zámek (semafor) uzamčen. Zámek je po ukončení kritické sekce uvolněn.

```
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.

Výstupy obou verzí programů jsou zobrazeny ve výpisu 4.33. Nejprve je dvakrát zobrazena náhodná hodnota bez použití zámku (*lockA.py*). Následuje použití zámku (*lockB.py*), kdy je výsledek sdílené proměnné 0, což je správná hodnota.

```
[]$ python3 lockA.py #bez zamku
Pocatecni hodnota sdilene promenne 0
Proces A ID: 3807
Proces B ID: 3808
Konecna hodnota sdilene promenne -106
---
[]$ python3 lockA.py #bez zamku
Pocatecni hodnota sdilene promenne 0
Proces A ID: 3921
Proces B ID: 3922
Konecna hodnota sdilene promenne -259
---
$[] python3 lockB.py #se zamkem
Pocatecni hodnota sdilene promenne 0
Proces A ID: 3781
Proces B ID: 3782
Konecna hodnota sdilene promenne 0
```

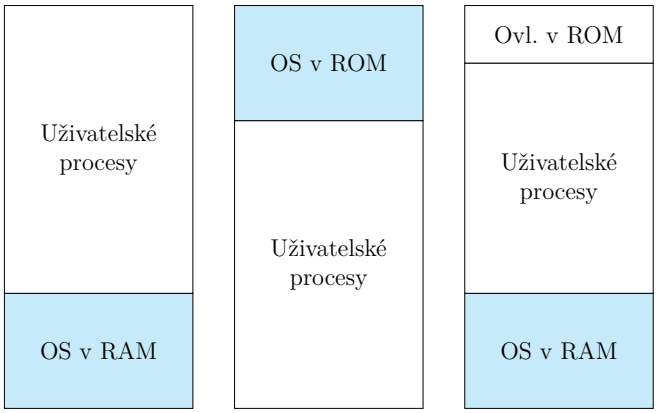
Výpis kódu 4.33: Hodnota sdílené proměnné bez a s použitím zámku.

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.

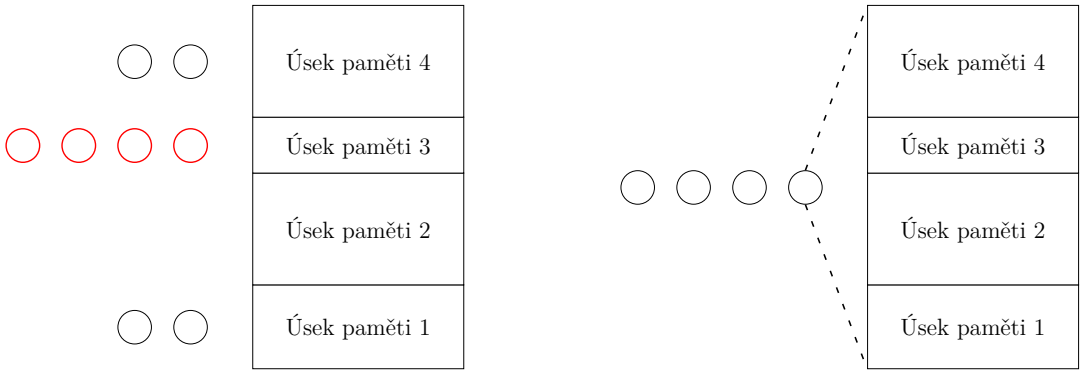
5.1 Dělení paměti

V paměti se může nacházet jeden nebo více uživatelských procesů. Pokud je v paměti jeden uživatelský proces, tak je styl práce následující: Uživatel zadá příkaz, operační systém přesune žádaný program do paměti a spustí jej. Jak program ukončí 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 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].

Pokud jsou procesy po 80 % svého času blokovány, tak by v paměti muselo být přinejmenším 10 procesů, aby vytížení procesoru bylo cca 90 %. Uvedený výpočet je pouze aproximací, jelikož uvažuje, že procesy jsou na sebe nezávislé. Procesy jsou však na sebe závislé. Pro přesnější výpočty se proto používá teorie front.

Jednoduchým způsobem 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 vejde. Tato situace je zobrazena na obrázku 5.2. Použit je úsek paměti, ve kterém bude nejmenší plýtvání paměti, jelikož neobsazené místo v tomto úseku nelze využít pro jiné procesy.

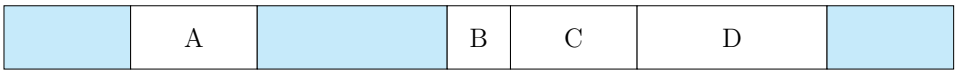


Obrázek 5.2: Statické rozdělení paměti – přidělování paměti procesům.

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 čekat ve frontě, i když jsou k dispozici úseky paměti o větší velikosti. Alternativou je použití jedné fronty pro všechny procesy. Pokud je nějaký úsek paměti volný, tak je přidělen prvnímu procesu ve frontě. Toto řešení přináší značné plýtvání paměti, protože procesy s malým uživatelským kontextem mohou být umístěny do velkých úseků paměti. Možná je modifikace, kdy je prohledána fronta s procesy a je vybrán proces s největším uživatelským kontextem, 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.

Při dynamické alokaci je procesu přidělena paměť podle velikosti uživatelského kontextu. Vznikají nesouvislé části volné paměti – **externí fragmentace**, jak je zobrazeno na obrázku 5.3. Volné části je možno sloučit do většího celku, který pak je k dispozici pro procesy s větším uživatelským kontextem. Pro tento úkon je typicky potřeba přesunout všechny procesy v paměti na nové umístění. V praxi se sloučení volné paměti nepoužívá z důvodu dlouhé doby provedení. Uvažme stroj s pamětí 256 MiB, který kopíruje 4 B za 40 ns. Přesun celé paměti zabere okolo 2,7 sekundy. Tuto akci je nutno provádět při každé alokaci nebo ve zvolených intervalech. Operační systém by byl zaneprázdněn slučováním volné paměti.

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



Obrázek 5.3: Volné úseky paměti při dynamické alokaci.

Procesy mění svou velikost – žádají o paměť a tu pak uvolňují (halda, zásobník). Mohou nastat tyto 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, které jeho rozšíření umožní. Toto je časově náročná operace.

Řešením je při alokaci paměti pro proces zabrat rezervu.

Virtuální paměť (VM – virtual memory) řeší situaci, kdy se všechny procesy nevejdou do hlavní paměti. Procesy jsou vyměňovány mezi **hlavní** a **odkládací** pamětí.

Virtuální paměť pracuje s:

- **Fyzickým adresovým prostorem (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 v hlavní paměti. Důvodem je vyšší rychlost přístupu.
- **Logickým adresovým prostorem (LAP)** (address space) – paměť z pohledu procesů. Je určen adresami, které je proces (operační systém) schopen generovat. LAP zahrnuje hlavní a odkládací paměť, viz obrázek 5.5.

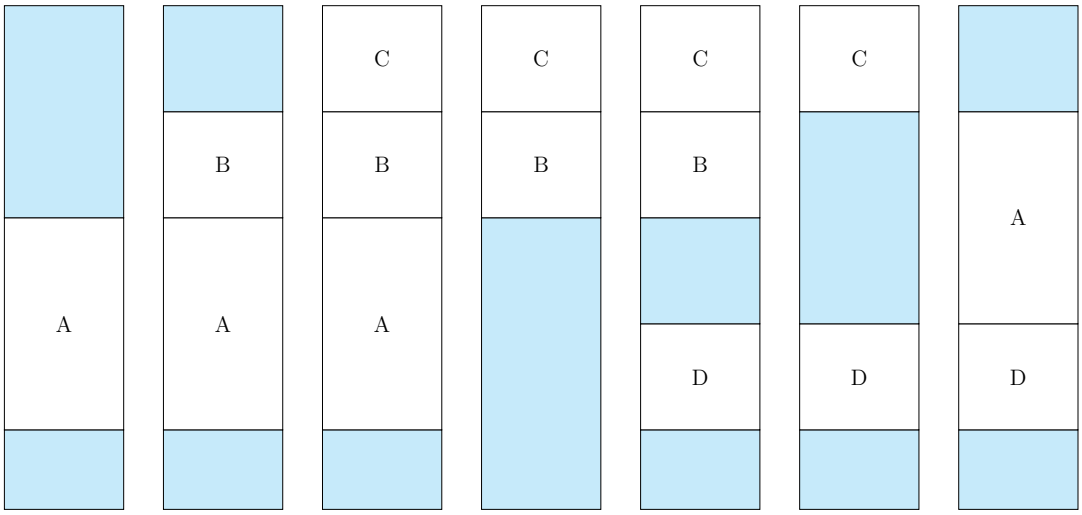
Například počítač s 32bitovým CPU, nabízí LAP o velikosti 4 GiB (2^{32}), ačkoliv fyzicky disponuje jedním 256 MiB paměťovým modulem (FAP).

Virtuální paměť lze realizovat i) vyměňováním celých procesů, ii) stránkováním, a iii) segmentací.

5.1.1 Vyměňování procesů

Celé procesy jsou 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 proces přesunut do odkládací paměti.

Příklad vyměňování celých procesů je uveden na obrázku 5.4 [11]. Nejprve je v hlavní paměti 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 jiném místě.



Obrázek 5.4: Příklad změn v hlavní paměti při vyměňování celých procesů.

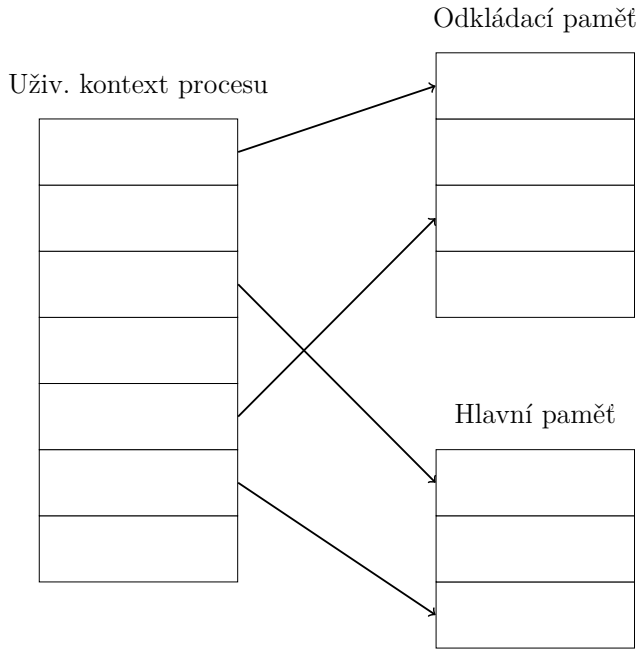
Umístění procesů v hlavní paměti se mění podle toho, jak jsou odkládány. Tím je opět způsobena externí fragmentace volné paměti. Vznikají volná místa v hlavní paměti, která nelze použít pro procesy s větší velikostí. Oblasti volné paměti je možné sloučit. Prakticky se však neprovádí pro časovou náročnost,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

Uživatelský kontext procesu lze rozdělit na části (overlays). Proces lze začít vykonávat, pokud má svou část v hlavní paměti. Další části procesu jsou přeneseny do hlavní paměti podle potřeby. V paměti lze umístit více částí procesu najednou. Historicky 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. Hlavní paměť má velikost 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 5.5.

[3]Uvedené hodnoty jsou ilustrativní.

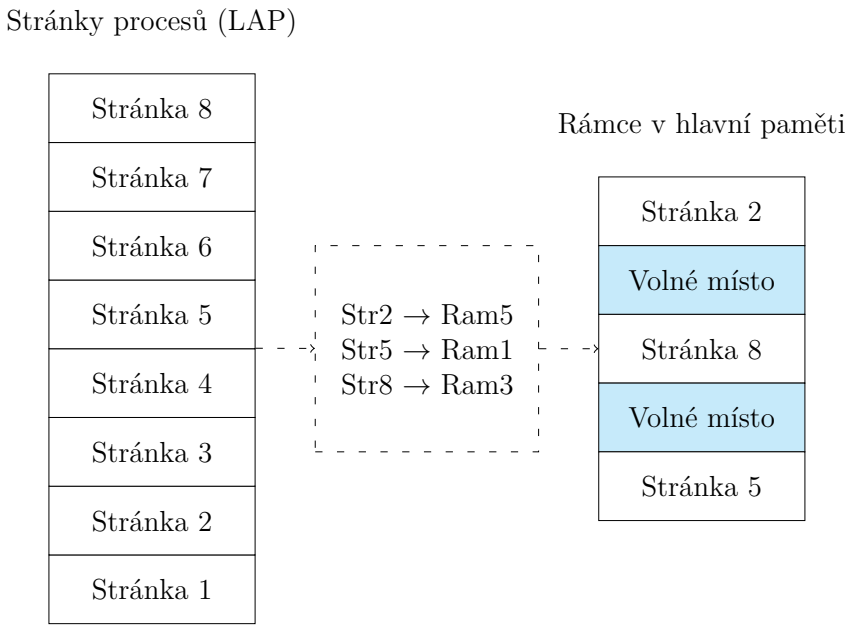


Obrázek 5.5: Koncept virtuální paměti.

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. Rámce a stránky 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 předešlý obsah přesunut do odkládací paměti. Výpadek stránky je pro proces neviditelný. Viditelný je pro operační systém, který vede evidenci umístění stránek v **tabulce stránek** (page table), která je zobrazena na obrázku 5.6 [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 5.6: Tabulka stránek.

Výhoda stránkování je, že stránky fixní velikosti (např. 4 KiB a 64 KiB) 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 důvody 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 uživatelský kontext 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 segmentů programů `ping` a `python` je uveden ve výpisu 5.1.

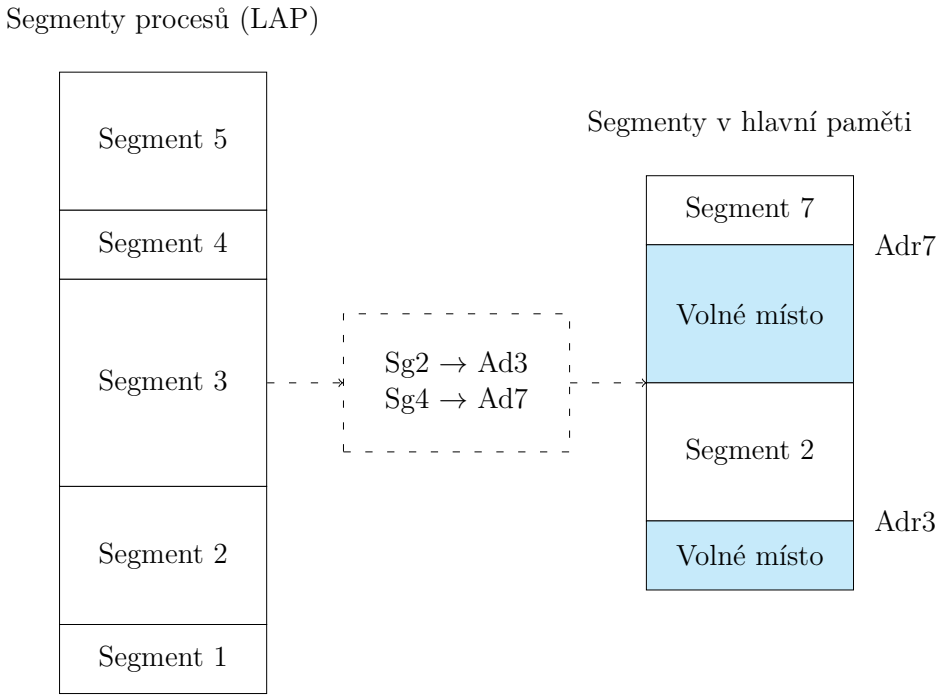
```
[]$ size /usr/bin/ping /usr/bin/python
text    data   bss      filename
55324   2440   145536   /usr/bin/ping
1859    628    4        /usr/bin/python
```

Výpis kódu 5.1: Příklad statických segmentů pro dva programy.

Bližší dělení segmentů pak například může být na:

- kódy funkcí,
- 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 5.7 [16].



Obrázek 5.7: 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 volné hlavní paměti.

Stránkování i segmentace mají své výhody i nevýhody, lze je kombinovat.

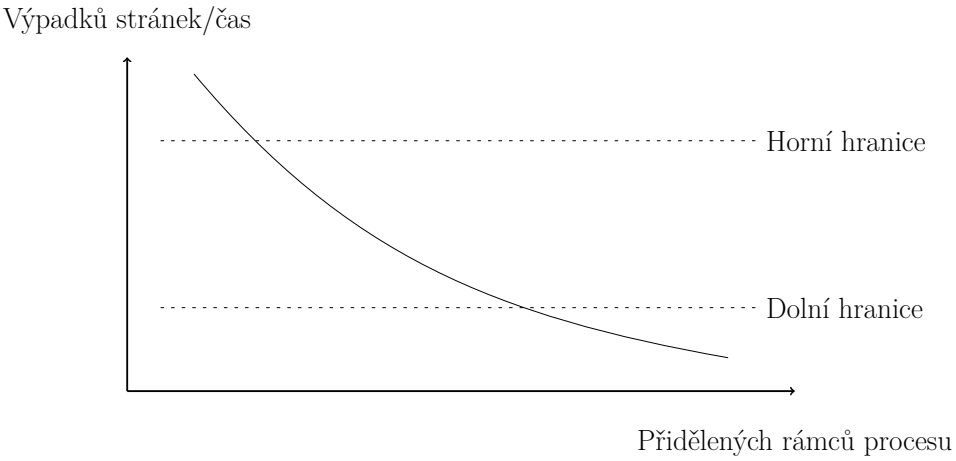
5.2 Přidělování paměti

Způsoby přidělení rámců v v hlavní paměti procesům jsou:

- Rovnoměrně – nerespektuje vlastnosti procesů.
- Podle velikosti – proporcionálně k velikosti uživatelského kontextu procesů (počtu stránek).
- Podle priority – priorita procesu se mění.
- Podle frekvence výpadků stránek – smyslem je vyhnout se stavu, kdy má proces mnoho výpadků stránek.

Stav, kdy pro má proces mnoho výpadků stránek, se nazývá **trashing** [16]. Alokace rámců je prováděna tak, aby frekvence výpadků stránek pro všechny procesy byla přibližně na stejné úrovni. Jestliže je frekvence výpadků pro proces velká, je to signál, že proces potřebuje více rámců; s větším počtem rámců přidělených procesu frekvence výpadků klesne. Pokud je frekvence výpadků stránek nízká, je to signál, že proces má příliš mnoho rámců. 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 5.8 [16]. Obrázek zobrazuje horní a spodní hranici, které jsou signály pro přidělení/odebrání hlavní paměti (rámců) danému procesu.



Obrázek 5.8: 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 odkládací 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 vybrané stránky z rámců do odkládací paměti. Tímto je stále udržován určitý počet volných rámců, které jsou ihned připraveny pro zaplnění stránkami.

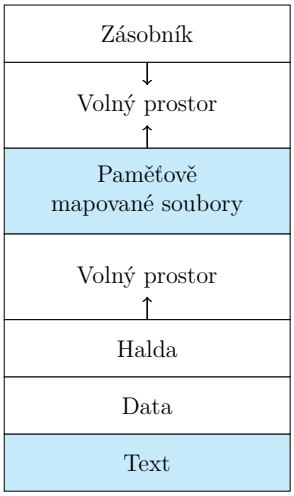
5.3.1 Odkládací paměť

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

- obyčejný soubor,
- odkládací soubor nebo odkládací oddíl.

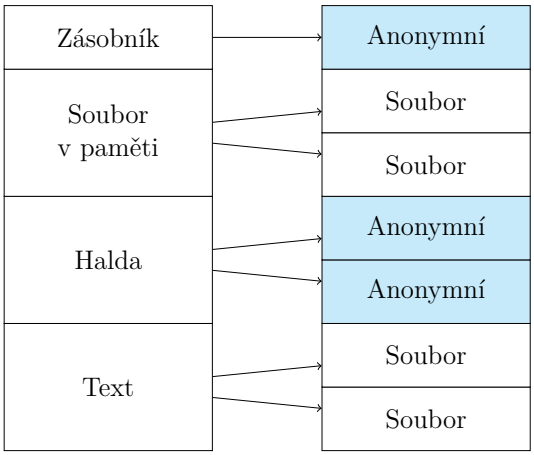
Stránky mohou být svázané s obyčejným souborem. Tento soubor může obsahovat programový kód procesu (data pouze pro čtení) nebo se může jednat o soubor připojený do uživatelského kontextu procesu, tzv. **paměťově mapovaný soubor**. Příklad rozložení uživatelského kontextu je zobrazen na obrázku 5.9. Soubor 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 funkce pro práci se soubory v souborových systémech. Výhodou je vyšší rychlost čtení a zápisu. Nevýhodou je 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. Paměťově mapovaný soubor lze sdílet mezi procesy.



Obrázek 5.9: Uživatelský kontext procesu svázaný s obyčejnými soubory (soubor s programovým kódem a paměťové mapovaný soubor).

Stránky, které nejsou svázané se souborem v souborovém systému, jsou nazývány **anonymní** a jsou přesunuty do **odkládacího souboru nebo oddílu** (swap file, swap partition)[5] na úložném zařízení. Příklad anonymních stránek je zobrazen na obrázku 5.10. Jedná se o části uživatelského kontextu data, haldy a zásobníku.



Obrázek 5.10: Uživatelský kontext a anonymní stránky.

[5]Záleží na implementaci podle operačního systému.

5.3.2 Algoritmy přesunu 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 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 dalších oblastí. Uveďme si příklad webového serveru (příklad je podán zjednodušeně bez uvažování práce OS). Server udržuje ve vyrovnávací paměti často zobrazované stránky. Když se paměť zaplní, tak se musí nějaká stránka z vyrovnávací paměti odstranit pro uložení nové stránky. Náhodně odstraněná stránka může být vzápětí znovu potřebná a uložena zpět do vyrovnávací paměti. Pokud je přesunuta webová stránka, která bude použita až za nějakou dobu, tak server bude pracovat rychleji.

Jakou stránku 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. Podívejme se na některá základní řešení, 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 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

[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.

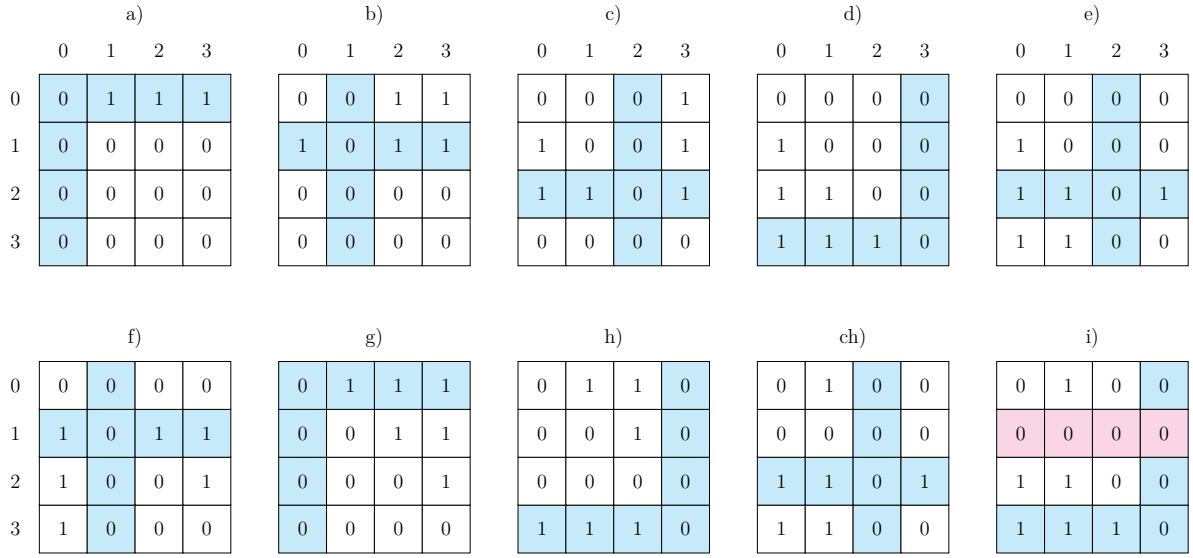
nepoužitá. Seznam je aktualizován při použití každé stránky. Nalezení stránky v seznamu a přesun na první místo je časově náročná operace.

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 5.11 [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 5.11: 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, např. 20 ms. Při každém intervalu je u každé stránky hodnota bitu A přidána k hodnotě programového čítače. Přidána je

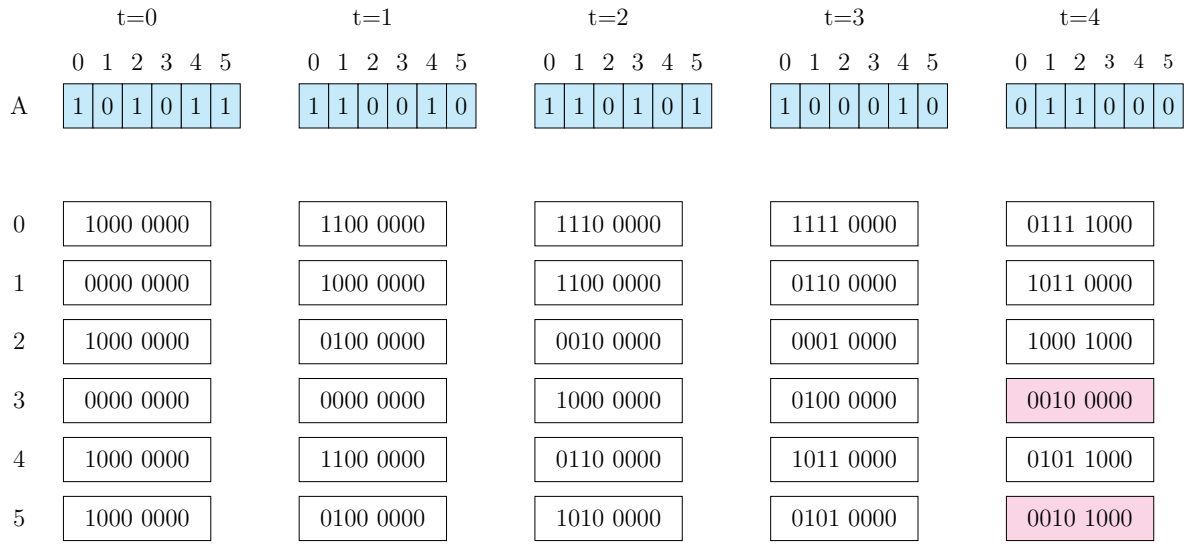
[7] Označováno také jako R (referenced).

hodnota 0, pokud ke stránce nebylo přistoupeno. 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].

Práci algoritmu lze pochopit z obrázku 5.12 [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 5.12: 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

Algoritmy řeší použití omezeného rychlého zdroje (hlavní paměť) a „neomezeného“ pomalého zdroje (odkládací paměť). 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 IP adresu ve formě země, města a souřadnic. Každý dotaz na polohu je za poplatek[9]. Při velkém počtu 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 vyrovnávací 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
2
3 #vypocet Fibonaciho cisla
4 def fib_without_cache(n):  #bez LRU
5  if n < 2:
```

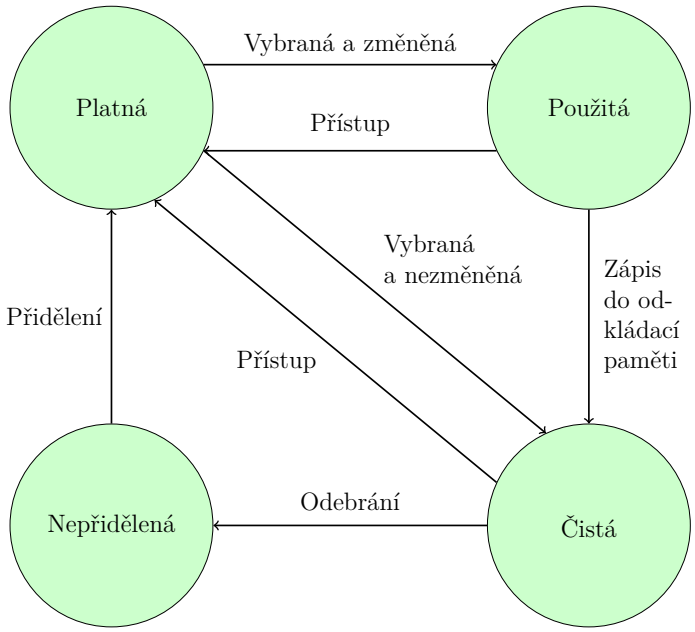
[9] Přibližně tisícinu dolaru a méně za dotaz, dle počtu požadovaných informací – poloha, ISP, AS, typ připojení atd.

```
  return n
 return fib_without_cache(n-1)+fib_without_cache(n-2)

begin = time.time()
fib_without_cache(45) #pocet iteraci 35(3s) 40(32s) 45(213s)
end = time.time()
print(end-begin) #cas vypoctu bez LRU
#213 s

@lru_cache(maxsize=128)
def fib_with_cache(n): #s LRU
 if n < 2:
  return n
 return fib_with_cache(n-1) + fib_with_cache(n-2)

begin = time.time()
fib_with_cache(45) #pocet iteraci 35(0s) 40(0s) 45(0s)
end = time.time()
print(end-begin) #cas vypoctu s LRU
#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 5.13 je zobrazen jednoduchý model stavů stránek [31]. Možné jsou tyto stavy:



Obrázek 5.13: Jednoduchý model stránek.

- 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“.

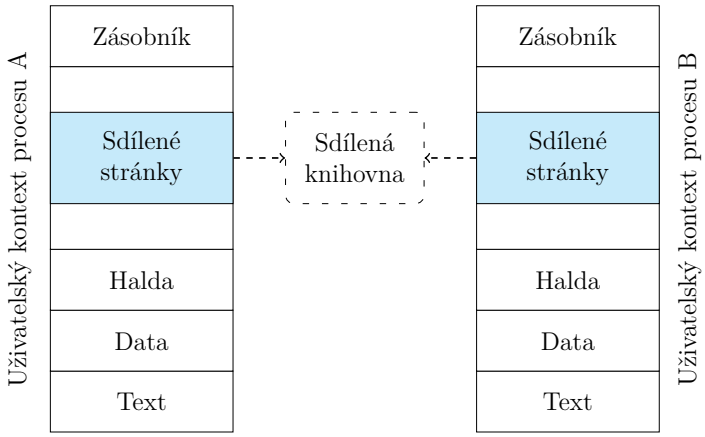
Stránky v hlavní paměti jsou periodicky prohledávány algoritmem (LRU, NFU, Aging) a jsou vybrány stránky k odložení pro uvolnění rámců hlavní paměti. Při vybrání stránky následují dvě možnosti:

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

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

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“. 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 s daty pro pouze čtení, např. knihoven, viz obrázek 5.14 [16]. Stránky sdílené knihovny jsou umístěny v části uživatelského kontextu pro paměťově mapované soubory.



Obrázek 5.14: 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 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. Proto operační systémy udržují seznam sdílených stránek.

5.5 Příklady na paměť

První příklad pracuje s odkládací pamětí na úložném zařízení, který slouží pro realizaci virtuální paměti. Na příkladu je popsáno, kde se odkládací paměť 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 paměť

Pro realizaci virtuální paměti se používá odkládací paměť, kam jsou přesunovány stránky. V operačním systému Linux je tato paměť realizována pomocí oddílu na paměťovém úložišti. V OS Windows je odkládací pamětí soubor. Který oddíl je využit jako odkládací lze zjistit příkazem `fdisk`[10], 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
/dev/sda1   83  Linux
```

[10] Je nutno právo administrátora

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

Výpis kódu 5.3: Odkládací paměť.

Přehled virtuální paměti lze získat příkazem `free`, viz výpis 5.4. Z výpisu je patrné, že odkládací paměť je využita 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 paměť lze využít i soubor v souborovém systému. Uveďme si příklad pro PC s 8 GB RAM a odkládací oddíl 8 GB (OS Linux) na kterém potřebujeme zpracovat velký soubor. 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 oddílu odkládací paměti. Postup 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 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 odkládací paměti.

Navýšení odkládací paměti (nyní oddíl i soubor) lze zjistit z 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 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 ve stránce, která se nenachází v hlavní paměti. Operační systém stránku přenese do hlavní paměti. Pokud v hlavní paměti není místo, systém vybere stránku a tu z hlavní paměti přesune do paměti odkládací. Operační systém může periodicky prohledávat stránky v hlavní paměti a přesunovat je do odkládací paměti. Takto 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 procesu. Dochází tak k značným přesunům mezi hlavní a odkládací pamětí. Informace o počtu výpadků stránek jsou zobrazeny ve 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. Jedná se i o stránky, které negenerují I/O operace – stránky jsou uloženy ve vyrovnávací paměti úložného zařízení.
- Položka *majflt/s* udává počet výpadků stránek, které jsou přesunuty z odkládací paměti – generují I/O operaci.

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

5.5.3 Rozložení paměti

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>

int main()
{
 int soubor = open("data.dat", O_RDONLY | O_CREAT);
```

```
 while(1);
}
```

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).

```
[]$ pmap -x PID_procesu
Address    Kbytes    RSS    Dirty    Mode    Mapping
0400            4      4        0    r-x--   pamet
0601            4      4        4    rw---   pamet
...
48ca           20     12       12    rw---   [ anon ]
48cf          136    108        0    r-x--   ld-2.17.so
4acf           12     12       12    rw---   [ anon ]
4aef            4      4        4    rw---   [ anon ]
...
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 odkládací paměti. 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 odkládací paměti, viz popis stavů stránek na obrázku 5.13.

Sloupec *Mapping* zobrazuje popisný název pro každou oblast. Oblasti paměti zahrnující programový kód a inicializovaná data jsou 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 není svázaná se souborem v souborovém systému. Anonymních stránky se odkládají do odkládacího souboru nebo oddílu.

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 paměťového prostoru procesu a následně je sváže s programem. U ostatních částí paměti procesu (nejsou svázány se souborem) je zobrazen popisek `[ anon ]`, tato oblast je tvořena anonymními stránkami.

Abychom zobrazili rozdíl mezi anonymními stránkami a stránkami, které jsou svázané se souborem, 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 odkládací paměti.

```
#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 je 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 odkládací paměti (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 je sdílený. 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);
```

```
4  mmap(0,1,PROT_READ,map_shared,soubor,0);
5  while(1);
6 }
```

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.

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

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

6 SOUBOROVÉ SYSTÉMY

Souborové systémy udržují metadata o uložených datech souborů. Tyto informace zahrnují **umístění** dat souborů na paměťovém médiu a **časové informace**, například čas posledního přístupu a poslední modifikace souboru. Dále udržují informace o **vlastnících** a přístupových **právech**.

Úložná zařízení jsou typicky rozdělena na několik oblastí, viz obrázek 6.1. **MBR** (Master Boot Record) obsahuje zavaděč, identifikátor disku a **tabulku oddílů** (partition table)[1]. Zavaděč v MBR je spouštěn programovým kódem BIOS (Basic Input-Output System). Zavaděč spustí zaváděcí (boot) sektor z aktivního oddílu (jeden oddíl v tabulce označen jako aktivní). Zaváděcí sektor startuje operační systém a je individuální pro daný operační systém. V tabulce oddílů jsou uloženy údaje o začátku a konci jednotlivých oddílů (partition) na úložném zařízení. Na oddílech se nachází souborové systémy.

Obrázek 6.1: Oddíly úložného zařízení.

6.1 Datové bloky a metadata

Souborové systémy ukládají soubory do **datových bloků**. Pro malý soubor je použit celý blok. V tomto bloku pak může zbýt volný prostor, který nelze použít pro data jiných souborů – **interní fragmentace**. Interní fragmentaci lze redukovat použitím menších bloků. Práce s každým blokem vyžaduje vyhledání jeho umístění v úložném zařízení, což může být časově náročná operace. Malá velikost datových bloků tedy přináší zpomalení operací se soubory. Souborové systémy mohou používat různé velikosti datových bloků. Velikost bloků je nastavena při formátování (vytváření) souborového systému na oddílu úložného zařízení.

6.1.1 Organizace dat na úložišti

Základní způsoby pro ukládání datových bloků jsou [11, 30]:

- kontinuálně – bloky souboru uloženy za sebou(contiguous allocation),
- nesouvisle – datové bloky obsahují ukazatele, které bloky svazují (linked allocation),
- nesouvisle – ukazatele na datové bloky jsou umístěny v indexačním bloku (indexed allocation).

[1] Volitelně může obsahovat i další části.

Ukládání bloků kontinuálně

Tento způsob ukládání je vhodný u plotnových disků. Načtení bloků celého souboru je rychlé, protože nevyžaduje přesun hlav (mimo přesuny hlav mezi cylindry, viz dále). Adresářový záznam obsahuje adresu prvního bloku souboru a velikost souboru (v podobě násobku počtu bloků).

Soubory jsou vytvářeny, mazány a modifikovány a dochází tím k dělení volného prostoru . Problém nastane, když největší souvislá část volného prostoru není dostatečná pro vykonání operace se souborem, i když celková velikost volného prostoru je dostačující. Historicky se problém fragmentace volného prostoru řešil tak, že všechna data byla naráz zkopírována z jednoho souborového systému na jiný souborový systém. Data v souborovém systému byla smazána a byl tak sjednocen volný prostor. Následně byla data opět zkopírována na původní souborový systém a byla tak uložena kontinuálně – byla provedena defragmentace. 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 ukazatel na první a poslední datový blok. Každý blok souboru obsahuje ukazatel na následující blok. Každý volný blok může být využit pro data souboru a nenastává tak problém externí fragmentace.

Problémem tohoto způsobu je čtení/zápis dat ze souboru. Pro načtení bloku souboru (např. data uprostřed souboru), je nutno nejdříve načíst první blok, zjistit ukazatel na následující blok, načíst následující blok atd. Každé načtení bloku, které mohou být rozmístěny kdekoliv na paměťovém médiu, značí časové zdržení (rotační disky). Z toho také plyne nevýhoda při poškození ukazatele (například závada paměťového média). Od poškozeného ukazatele dále dojde ke ztrátě dat souboru. Další nevýhoda je velikost ukazatele. Pokud je například velikost ukazatele 4 B a velikost bloku 512 B, tak 0,78 % každého bloku nemůže být použito pro data souboru. Částečným řešením je sdružit bloky do skupin. Ukazatel na skupinu bloků zabírá méně místa ve skupině. Zvyšuje se také rychlost práce s daty. Nevýhodou je interní fragmentace, která nastává, když ukládaná data mají menší velikost, než je velikost skupiny bloků. Nevyužitý prostor skupiny bloků nelze použít pro data jiného souboru.

Ukládání bloků nesouvisle – ukazatele v indexačním bloku

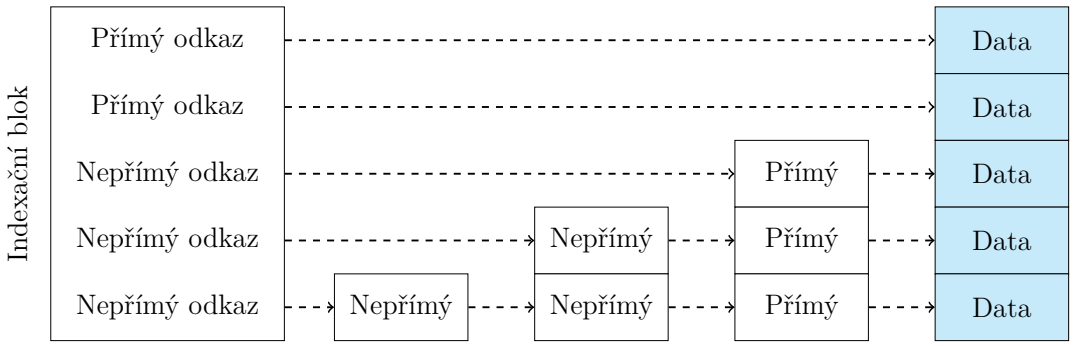
Tento způsob ukládá ukazatele na datové bloky souboru na zvláštním místě – v indexačním bloku. Každý soubor má svůj vlastní indexační blok, který obsahuje seznam ukazatelů na bloky souboru s vlastními daty. Adresářový záznam obsahuje ukazatel na indexační blok souboru. Toto řešení umožňuje přímé načtení zvoleného datového bloku (například data uprostřed souboru). Při zápisu dat do volného datového bloku je provedena aktualizace ukazatele v indexačním bloku.

Použití indexačního bloku nezpůsobuje externí fragmentaci. Nevýhoda je v zabírání více místa pro ukazatele na datové bloky než je nutné. Příkladem může být malý soubor

o velikosti dvou bloků. Zde bude zabrán celý indexační blok (např. 512 B), a to pouze pro dva ukazatele (např. 4 B).

Používají se tyto přístupy pro indexační bloky:

- Indexační blok obsahuje ukazatele na datové bloky – **přímé odkazy**. V případě velkého souboru je propojeno více indexačních bloků.
- Indexační blok obsahuje ukazatele na další indexační bloky – **nepřímé odkazy**. Vzniká hierarchické uspořádání s několika úrovněmi. Na poslední úrovni je indexační blok s přímými odkazy.
- Kombinace obou přístupů, viz obrázek 6.2 [11]. Indexační blok obsahuje přímé i nepřímé odkazy. Výhoda je v tom, že malé soubory lze pokrýt jedním indexačním blokem pomocí přímých odkazů. Pro velké soubory je využito nepřímých odkazů s několika úrovněmi.



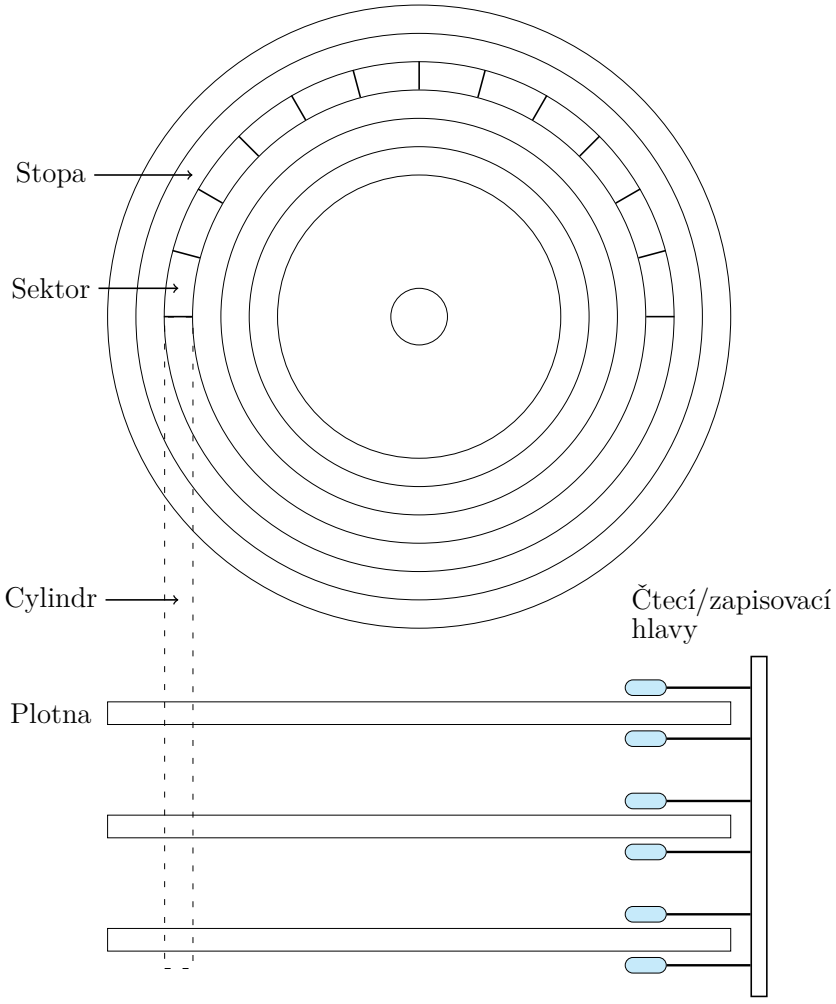
Obrázek 6.2: Příklad kombinace přímých a nepřímých odkazů na datové bloky.

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 obsahovat až 256 ukazatelů na jiné bloky. Při použití 10 přímých odkazů, 1 bloku pro nepřímou indexaci první úrovně, 1 bloku pro nepřímou indexaci druhé úrovně a 1 bloku pro nepřímou indexaci třetí úrovně je maximální velikost souboru 16 GiB podle výpočtu $10 \times 1\,\text{KiB} + 256 \times 1\,\text{KiB} + 256 \times 256 \times 1\,\text{KiB} + 256 \times 256 \times 256 \times 1\,\text{KiB} = 16\,\text{GiB}$.

6.1.2 Ukládání dat a metadat

Data jsou ukládána v úloženém zařízení v podobě sektorů. Sektory rozlišujeme fyzické a logické. Fyzický sektor může mít stejnou nebo větší velikost (Advanced Format) než logický. Velikost fyzického sektoru udává nedělitelnou jednotku dat 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 nejdříve načten fyzický sektor. Následně je aktualizován jeho logický sektor a pak uložen celý fyzický sektor. Datové bloky definované souborovým systémem mohou mít velikost jednoho nebo více sektorů.

Problémem pevných disků s plotnami 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 se soubory je potřeba nejdříve načíst metadata a z nich vyčíst informace o pozici datových bloků souboru a případně přístupová práva. 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 daty souboru. Přesun hlav mezi metadaty a daty přináší časová zdržení. Z tohoto důvodu jsou plotnové disky rozděleny na „samosprávné“ oblasti zvané **cylindry**, které obsahují data a metadata souboru, viz obrázek 6.3. Cylindr je sada stop, které jsou umístěny nad sebou na plotnách disku. Cylindrická skupina zahrnuje několik cylindrů a obsahuje metadata i data. Nedochází tak k nadměrným přesunům hlav při práci s metadaty a 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. Důvodem je redukovaný pohyb hlav na stopami plotny. Rychlost práce s daty je také dána rychlostí otáčení ploten. Po přesunu hlavy na stopu musí ještě plotna dorotovat na daný sektor. Pro zvýšení rychlosti práce se provádí defragmentace obsahu souborů, tj. uložení bloků souboru souvisle.

U paměťových médií typu SSD (Solid State Drive) nenastává výše uvedený problém. Z toho důvodu se zde defragmentace neprovádí. Dalším důvodem je skutečnost, že počet zápisů u tohoto média je technologicky limitován. Překročení limitu může způsobit nefunkčnost média. Defragmentace je zde tedy nežádoucí, jelikož navyšuje počet zápisů. Ovladač SSD média může provádět zápis dat tak, aby celkový počet zápisů byl rozložen rovnoměrně nad celým médiem.

Příklad použití znalostí o ukládání metadat a dat je obnova smazaných souborů na paměťovém úložišti. Zde rozlišujeme obnovu pomocí opětovného získání metadat, tedy informací souborového systému, např. ze zálohy metadat na paměťovém úložišti. Pokud tyto pokusy selžou, lze soubory vyčítat „surovou“ formou z datových bloků úložiště. Zde ovšem dochází k částečné nebo plné ztrátě názvu souboru. Adresářové struktura je ztracena zcela. Informace o souborech (např. typ) je brána z počátečních vzorů dat přímo v souboru.

6.1.3 Konzistence dat a metadat

Operace v souborových systémech (zápis, mazání) mohou být přerušeny z důvodu výpadku napájení, systémového pádu atd. To může vést k zanechání dat v **nekonzistentním stavu**. Příklady situací, které mohou nastat jsou:

- Soubor nebyl smazán, i když by požadavek k jeho smazání zpracován.
- Část dat v souboru byla aktualizována, další část již ne.

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

1. odstranění záznamu o souboru z adresáře,
2. označení i-uzlu [2] souboru jako volný,
3. označení datových bloků jako volných.

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

Pro předcházení vzniku nekonzistentních dat se používají **žurnálovací souborové systémy** (journal file systems) [32, 16]. Tyto souborové systémy udržují v „žurnálu“ plánované operace s daty před jejich vlastním provedením. V případě pádu systému je provedena obnova dat pomocí tohoto plánu. Operace jsou provedeny znovu do té fáze, kdy jsou data a metadata konzistentní. Každá operace v žurnálu je proveditelná pouze

[2] 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.

jako celek, tj. provedená kompletně před pádem systému, nebo znovu 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 nedokončeného zápisu bude celý záznam ignorován. Operace v žurnálu mohou být udržovány v podobě cyklického logu s danou velikostí. V případě úspěšné operace je záznam v žurnálu smazán.

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

- Ukládána pouze metadata. V tomto případě může nastat situace inkonzistence dat. Příkladem může být přidání dat do souboru. Nejprve je změněn záznam o velikosti souboru (metadata). Dále jsou vybrány volné datové bloky (metadata). Posledním krokem je, že do volných bloků jsou zapsána nová data souboru (data). Při ukládání pouze metadat do žurnálu nebude pokryta poslední operace. V případě pádu bude tedy soubor rozšířen o data, kterými bude předchozí obsah v volných datových blocích (náhodná data).
- Ukládána metadata i data. Do žurnálu jsou ukládány kopie dat, které se následně budou zapisovat do souboru. V případě pádu systému, je zápis do datového bloku proveden znovu pomocí kopie dat v žurnálu. Z důvodu vytváření kopie dat je zápis dat pomalý. Používá se v systémech, kde je vyžadována kompletní ochrana dat.

Pro zvýšení spolehlivosti lze zálohu žurnálu umístit na sekundární úložné zařízení z důvodu možné poruchy primárního. Žurnál také může být ukládán na jiném úložném zařízení než data z důvodu vyšší rychlosti zápisu. Nedokončené operace zaznamenané v žurnálu jsou provedeny při následujícím připojení souborového systému.

6.1.4 Virtuální soubory

Mimo „reálné“ souborové systémy, které ukládají reálná data do souborů, se v operačních systémech používají **virtuální souborové systémy** [16] s virtálními soubory[3]. Obsah virtuálním souborů se může generovat dynamicky teprve na základě požadavku, tj. data souboru před požadavkem neexistovala. Virtuální soubory se například používají pro zobrazení informací o jádře[4].

Virtuální souborové systémy mohou obsahovat soubory, které reprezentují zařízení. Tyto speciální soubory umožňují přímý přístup k zařízení tak, že 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 fyzických zařízení se mohou soubory vytvářet automaticky a reflektovat tak změny v aktuálně dostupných zařízeních. Rozlišujeme dva druhy zařízení,

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

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

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

Tabulka 6.1: Dělení adresářů pro jejich organizaci.

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.
- 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 systémy obsahují velké množství souborů. Proto je nutné je organizovat. Existují různé varianty organizace, které mohou být specifické pro jeden operační systém, nebo může být organizace stejná pro více operačních systémů. Rozšířený je standard **FHS** (Filesystem Hierarchy Standard) [35]. Jeho znalost dává možnost „předvídat“, kde se soubor či adresář nachází v operačních systémech, které tento standard používají. Základem je dělení souborů na:

- Statické (nemění svůj obsah) a dynamické (mění svůj obsah). Důvodem je uložení statických souborů na médium s přístupem pouze pro čtení nebo připojení souborového systému pouze pro čtení. U dynamických souborů může být v pravidelných intervalech prováděna jejich záloha.
- Nesdílitelné (specifické pro jedno zařízení) a sdílitelné (použitelné na více zařízeních). Důvodem je sdílení adresářů po síti.

Příklad organizace je uveden v tabulce 6.1. Statické soubory jsou v adresářích `/usr` a `/opt`. Tyto soubory lze sdílet, jelikož jsou použitelné na více zařízeních. Statické soubory v adresářích `/boot` a `/etc` jsou specifické pro danou stanici (např. startovací menu a konfigurace služeb), a tudíž jsou nesdílitelné.

V adresáři s dynamickými daty `/var` je podadresář `mail`, kde jsou uloženy mailové schránky uživatelů. Mailové schrány uživatelů lze sdílet. Dále jsou uvedeny podadresáře `lock` a `log`. V adresáři `lock` jsou ukládány „zamykací“ soubory pro řízení přístupu; prezence zamykacího souboru značí aktuální používání prostředku. V adresáři `log` jsou záznamy činnosti systému a aplikací. Tyto soubory jsou platné jen pro dané zařízení a nelze je sdílet.

6.2.1 Primární organizace

Standard definuje povinné a volitelné adresáře na první úrovni (v kořenovém adresáři `/`). 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 realizujícími 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[5]. Příkladem je binární soubor s jádrem `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`. První speciální soubor generuje náhodná čísla a druhý nuly.
- `/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.
- `/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 úložném zařízení (FLASH disk).
- `/mnt` – Místo pro dočasně připojované souborové systémy. Příkladem je soubor reprezentující dočasně připojený síťový souborový systém.
- `/opt` – Obsahuje instalované programy[6], každému typicky odpovídá vlastní podadresář.
- `/sbin` – Obsahuje systémové nástroje pro správu operačního systému. Jejich použití je typicky omezeno pro administrátory. 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.

[5] Další varianta pro umístění jádra je přímo v kořenovém adresáři.
[6] Instalované programy se umísťují na několik míst, toto je jedno z možných.

- `/usr` – Další hierarchie adresářů. Jedná se o sdílitelná data, která mohou být dostupná pouze ke čtení. Příkladem je adresář `src` se zdrojovými kódy. Bližší popis této sekundární organizace adresářů je uveden dále.
- `/var` – Obsahuje dynamická data. Zahrnuje podadresáře s logy programů/celého systému a data webových stránek. Povinné podadresáře jsou `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 uživatelského účtu.
- `/root` – Domovský adresář uživatele *root*.

6.2.2 Sekundární organizace

Povinný adresář `/usr` tvoří druhou hierarchickou strukturu. Tento adresář obsahuje sdílitelná data pouze pro čtení. Adresář 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++.
- `local` – Zde je možné instalovat programy.
- `sbin` – Obsahuje binární soubory s programy realizující příkazy pro správce systému, které nejsou zásadní pro operační systém (programy nutné pro správu systému jsou umístěny v adresáři `/bin`). 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 příkazů. Bývají zde také umístěny slovníky, například se slovy anglického jazyka[7].

V adresáři `/usr` se mohou nacházet volitelné podadresáře. Příkladem je adresář `include`, který obsahuje hlavičkové soubory pro programy v jazyce C++, a `src`, který obsahuje zdrojové kódy.

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.

[7] Je zde také umístěn slovník využitý v počítačovém cvičení, které se zabývá prolomením hesla.

6.3.1 Podporované souborové systémy

Operační systémy typicky podporují více souborových systémů pomocí ovladačů. V OS Linux jsou ovladače dostupné jako moduly jádra. Pokud je požadavek na využití souborového systému, tak je do jádra nahrán příslušný modul.

Informace o využívaných souborových systémech jsou v souboru `/proc/filesystems`, viz výpis 6.1. První sloupec ve výpisu označuje, zda je souborový systém připojený k blokovému zařízení. Text *nodev* značí, že souborový systém není připojený k reálnému zařízení – jedná se o virtuální souborový systém. 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.

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 USB zařízení v podobě souborů. Pomocí těchto souborů jsou pak USB zařízení dostupná. Souborový systém `proc`, který je připojen do adresáře `/proc`, zobrazuje vnitřní informace o systému. Jedná se například o údaje o paměti a procesech. Tyto informace nejsou v souborech předem uloženy. Pokaždé, když se k informacím přistupuje, jsou teprve data do souboru generována. Souborový systém `pipefs` je použit pro komunikaci pomocí „rour“ (pipes) `|`. Souborový systém `sockfs` slouží pro síťovou komunikaci. Data jsou přenášena po síti pomocí zápisu/čtení ze souborů svázaných se sokety.

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. Plotny jsou rozděleny na stopy a sektory. 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 nejdříve nutno načíst metadata, zjistit polohu vlastních dat a pak přesunout hlavy pro přístup k datům. Vzdálenost mezi metadaty a daty způsobuje zpoždění při práci s daty. Tento problém se řeší seskupením stop do cylindrických skupin. Každá cylindrická skupina obsahuje k vlastním datům i metadata, která jsou umístěna tak, aby nedocházelo k nadměrným přesunům hlav.

```
[]# 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: Úložná 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 připojený oddíl. Následující položka označuje souborový systém. 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. Přehled o stavu volného místa na 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

Metadata jsou ukládána ve struktuře i-uzel (i-node). I-uzel obsahuje typu souboru, vlastník, skupina vlastníka, přístupová práva, velikost souboru, čas vytvoření, čas posledního čtení a modifikace. Dále obsahuje ukazatele na datové bloky souboru. Malé soubory jsou adresovány pomocí přímých odkazů. Velké soubory jsou adresovány pomocí nepřímých odkazů.

Adresáře obsahují jméno souboru a číslo jeho i-uzlu. V různých adresářích může být stejné jméno, které ukazuje na jeden soubor. Také může být ve stejném adresáři více jmen, která ukazují na stejný soubor. Tyto jména označujeme jako pevné odkazy (hard links). Pokud je smazán pevný odkaz na soubor, tak počet pevných odkazů na soubor je snížen o 1. Pokud je počet pevných odkazů na soubor roven 0, tak je soubor smazán. Dalším typem odkazu je symbolický odkaz. Zde 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 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 dvě výchozí položky. První z nich je „.“ (tečka), která odpovídá i-uzlu samotného adresáře[8]. 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. Čtením dat ze souboru čteme data ze zařízení, zápisem do souboru toto zařízení ovládáme (ukládáme data). Jsou tedy použity

[8] Je to odkaz sám na sebe.

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 pomocí příkazu `ls -l`:

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

Seznam speciálních souborů lze získat výpisem adresáře `/dev`, kde je připojen virtuální souborový systém, viz výpis 6.7. V tomto souborovém systému mají zařízení odpovídající soubor (nebo části zařízení v případě oddílů na paměťovém úložišti):

- `/dev/sda` – celé úložné zařízení,
- `/dev/sda1` – první oddíl,
- `/dev/sda2` – druhý oddíl,
- `/dev/null` – softwarové zařízení; slouží jako „černá díra“, data zapsaná do zařízení jsou ztracena; typické použití je zahazování hlášení programu tištěných do terminálu,
- `/dev/zero` – softwarové zařízení; slouží jako generátor nulových bajtů; využitelné například pro přepisování předchozích dat,
- `/dev/urandom` – 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  urandom
```

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

Speciální soubor `/dev/urandom` lze například využít pro generování hesel. Data ze speciálního souboru budou čtena příkazem `cat`. Jelikož čtení znaků příkazem `cat` by bylo nekonečné (protože náhodná čísla jsou generována nekonečně), 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 jsou převedeny příkazem `tr` na písmena „a“ až „z“ a to velká i malá. Příkaz `tr` s parametry ve výpisu 6.9 provádí filtraci znaků, tak že jsou ponechány jen znaky ze zadané množiny.
2. Počet písmen hesla je omezen příkazem `fold`, který provádí seskupování znaků.
3. Počet hesel je definován příkazem `head`, který specifikuje počet řádků k zobrazení. Každý řádek odpovídá jednomu heslu.

Ve výpisu 6.9 jsou vygenerovaná náhodná hesla. Parametrem příkazu `tr` je množina možných znaků 'A-Za-z', které jsou předány dále ze vstupu (jiné jsou zahozeny). Pokud bychom chtěli například generovat PIN, tak by parametrem příkazu bylo '0-9' (jiné znaky jsou 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. generuje se pět hesel.

```
[]$ 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.

Tato hesla lze využít v situaci, kdybychom zaváděli nového uživatele do systému, nebo skupinu uživatelů. Vygenerovaná hesla jsou nastavena pro vytvořené účty pomocí příkazu `passwd`. Uživateli je heslo následně předáno. Uživatel pak může (by měl) toto heslo změnit za své vlastní.

7 SÍŤOVÝ SYSTÉM

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

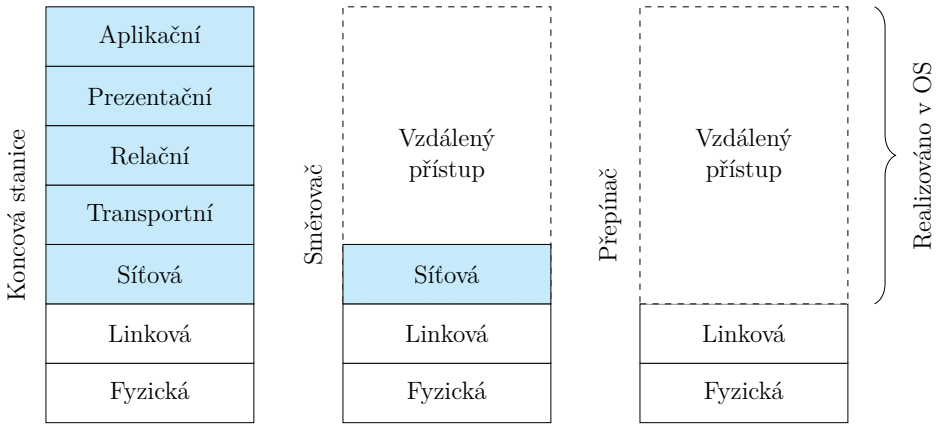
Z pohledu komunikace operační systém realizuje vrstvy od síťové až po aplikační dle referenčního modelu OSI (model není vázán na konkrétní protokoly). Tyto vrstvy jsou zachyceny na obrázku 7.1 (barevně označeny u koncové stanice). Operační systém tedy zajišťuje (podle definice jednotlivých vrstev):

- Rozšíření o síťovou část – aplikační vrstva.

[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.

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

- **Systémové porty** (0–1023). Jsou rezervovány pro známé síťové služby (např. port 69 pro TFTP). Označují se také jako známé nebo privilegované porty, protože tyto

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

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.

```
[]$ 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

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

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.

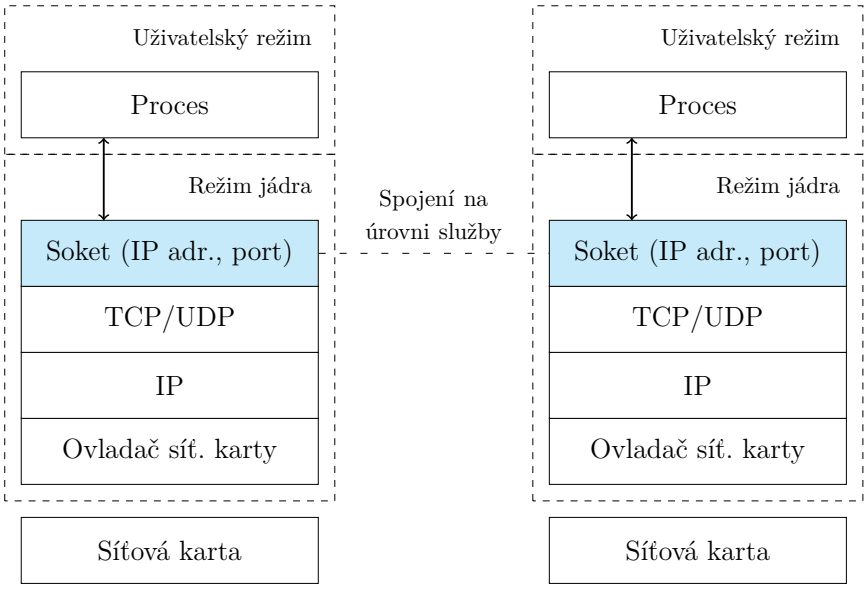
7.2.1 Stavy komunikace

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

1. čekání na spojení,
2. sestavování spojení,
3. přenos dat,
4. ukončování spojení.

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

[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.

Přechody mezi stavy jsou řízeny pomocí **příznaků** v záhlaví zasílaných segmentů protokolu TCP. Základními příznaky pro řízení stavu soketů jsou:

- ACK – potvrzení přijatých dat.
- PSH – přenos dat.
- SYN – žádost o sestavení spojení.
- FIN – ukončení spojení.

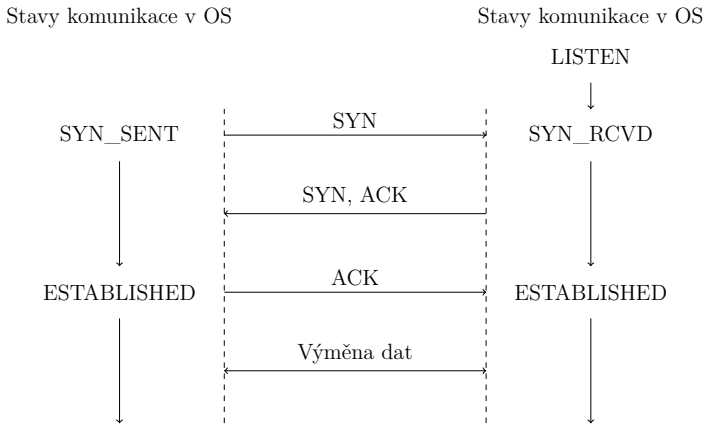
Pro sestavení spojení jsou použity příznaky SYN a ACK. Pro ukončení spojení jsou použity příznaky FIN a ACK. Pro přenos dat se používají příznaky PSH a ACK.

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 7.3.

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



Obrázek 7.3: Stavy komunikace v operačním systému – sestavení spojení.

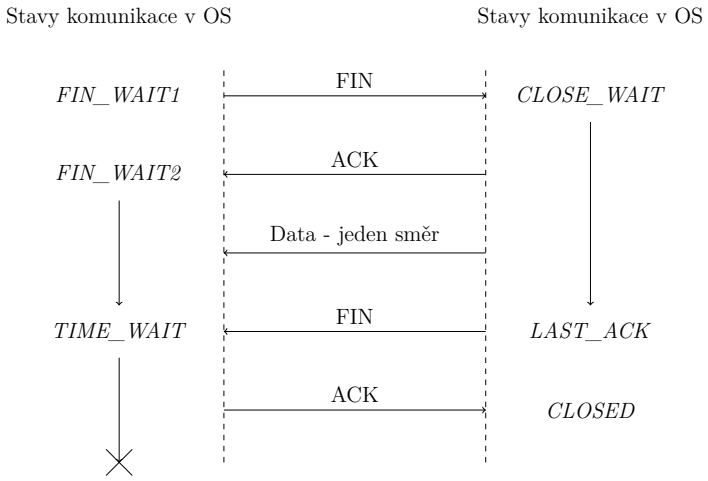
Při sestavování spojení může být soket na straně serveru ve stavu LISTEN (poslouchá/čeká na příchozí spojení) nebo SYN_RCVD (přijal od klienta první segment SYN). Soket na straně klienta se může nacházet ve stavu SYN_SENT (poslán segment SYN). Po navázání spojení přejdou oba sokety do stavu ESTABLISHED.

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, 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 7.4.



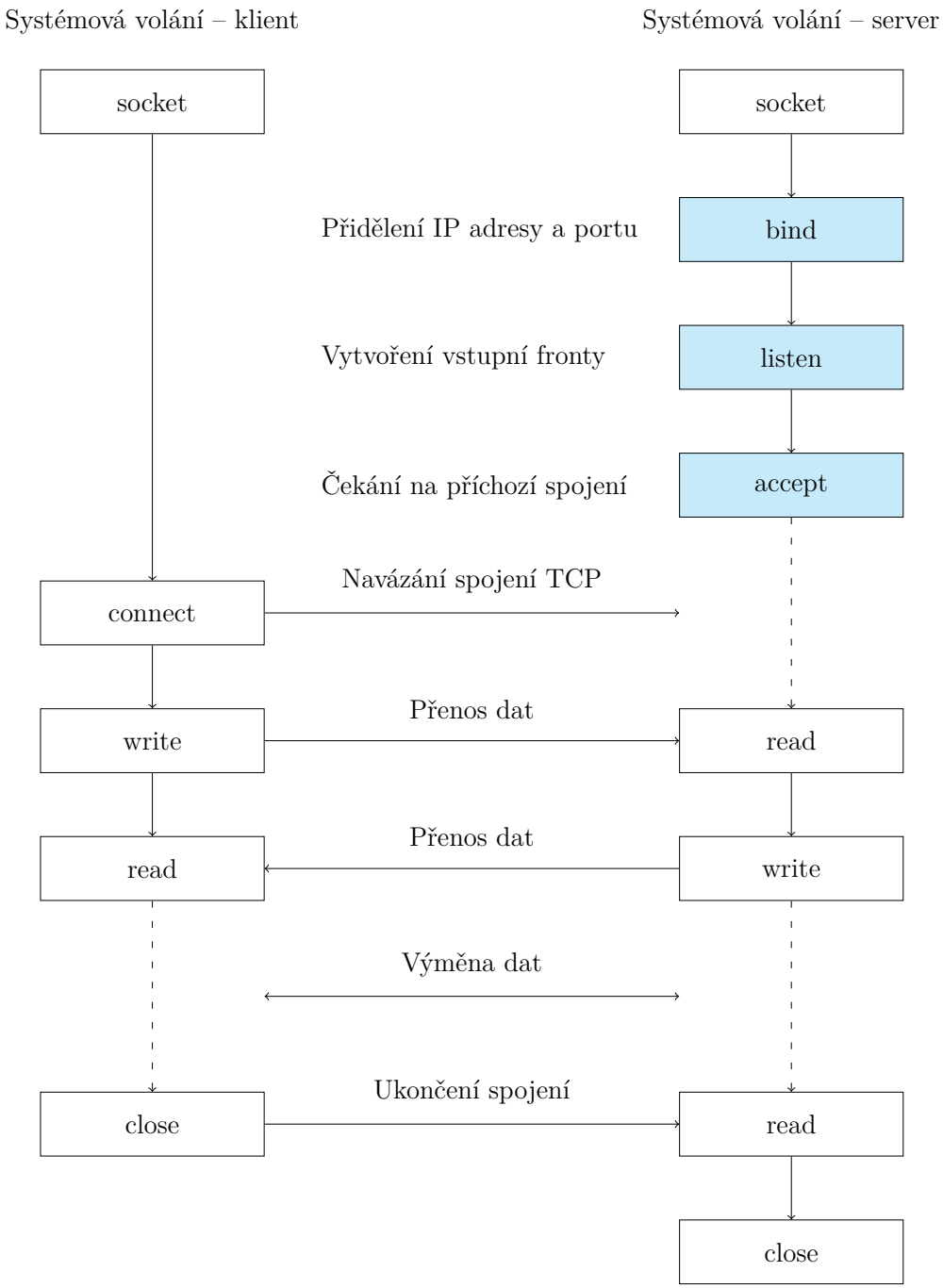
Obrázek 7.4: Stavy komunikace v operačním systému – ukončení spojení.

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 7.5.



Obrázek 7.5: 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

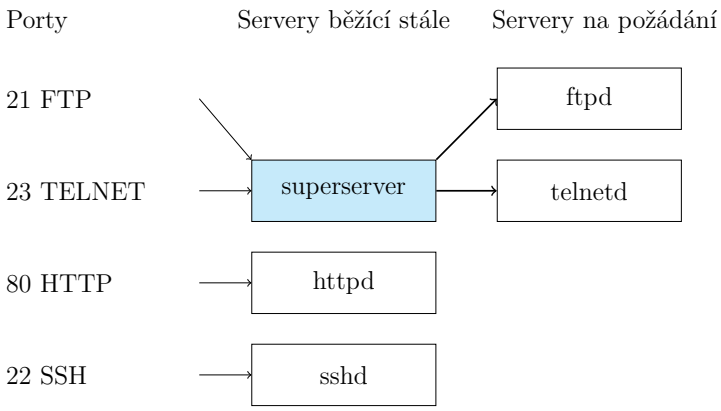
Služby operačního systému jsou zprostředkovány programy, které nazýváme servery. Slovo server používáme obecně pro „entitu“, která poskytuje síťovou službu (například server SSH či TELNET). Proces každého serveru má přiřazenu adresu na transportní vrstvě – port. Tato adresa rozlišuje jednotlivé servery 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 7.6. 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 7.6: 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.

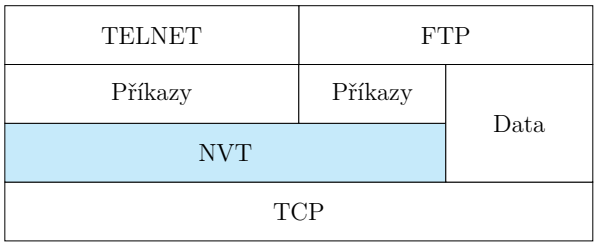
```
service telnet
{
user            = root
server          = /usr/sbin/in.telnetd
disable         = yes
}
```

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 7.7.



Obrázek 7.7: 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:

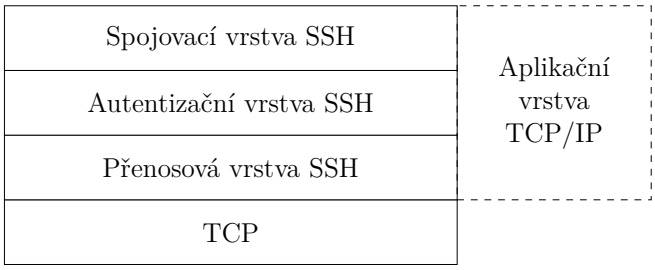
- Autentizace – ověření **identity** (ověření, zda je uživatel/systém tím, za koho se vydává). Každé spojení přes SSH vyžaduje dvě autentizace. Klient ověřuje identitu serveru a naopak server ověřuje identitu uživatele.
- Integrita – neměnnost dat při přenosu přes síť. Pokud třetí entita vstoupí do relace 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 7.8 [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 7.8: 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 autorizováni.
- 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.

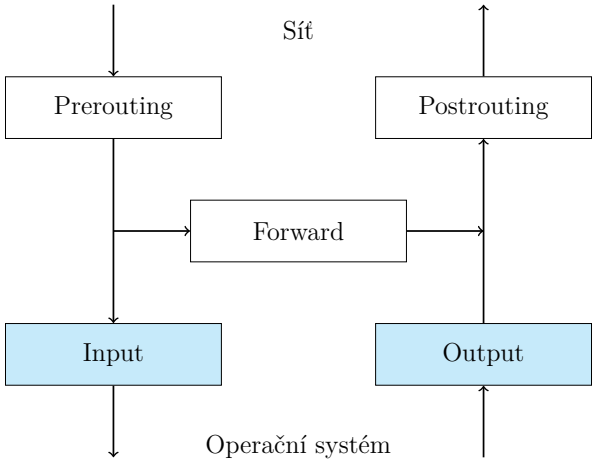
Firewally dělíme dle úrovně filtrování na paketové, stavové a aplikační [15, 25].

- 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 7.9. Jednotlivé části představují řetězce (bloky) pravidel, kterými jsou:



Obrázek 7.9: 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].

```
1 []$ nmap -sn -PS22 localhost
```

[9] http://nmap.org/

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

```
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.

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 (Maximum Transmission Unit) 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.

7.5.3 Získání obsahu komunikace

Data jsou zachycena pomocí nástroje v textovém režimu. Práce v textovém režimu má své opodstatnění, protože monitorování provozu je prováděno na síťových prvcích, které nemají grafické rozhraní. Pro reálné využití je potřeba mít přístup na 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 komunikace mimo lokální síť.

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.

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

[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 operačních systémů. 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 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.

Na závěr skript 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.