KAPITOLA 3

# Základy vývoje pro jádro

Následující odstavce přinesou základní informace o tom, jak vůbec vyvíjet součásti jádra. I když je většina postupů stejná nebo velmi podobná jako u uživatelských programů, některé důležité věci se liší. Hlavně je důležité při psaní pro jádro přísně dodržovat základní pravidla – laxní přístup se může krutě vymstít.

## Základní pravidla

Jak už bylo řečeno v odstavcích o omezených možnostech v jádře, platí při vývoji poměrně striktní pravidla, která je nutno bezpodmínečně dodržovat. Jinak na sebe problémy nenechají dlouho čekat – v lepším případě se projeví „jen" zpomalením systému, v horším záludnými zátuhy nebo dokonce poškozením dat.

Podívejme se proto na pravidla, která by měl každý vývojář vzít za svá. Skoro by se slušelo nazývat je patnácterem přikázání.

1. **Předem kvalitní rozhraní navrhneš.** Ještě dříve, než je napsáno první písmeno kódu modulu, je potřeba navrhnout jeho vnější rozhraní. Občas je samozřejmě potřeba rozhraní později změnit (při nových požadavcích apod.), ale i s tím by se mělo předem počítat. Dobré je v co největší míře využívat existující rozhraní (standardní systémová volání apod.), a to vždy takovým způsobem, jak to bude uživatel přirozeně očekávat.
2. **Svůj kód čistý a přehledný udržovati budeš.** V chaotickém a nepřehledném kódu se špatně hledají chyby, těžko odhaduje význam posloupností příkazů atd. Kód by měl být psán vždy srozumitelně, i když to třeba bude vyžadovat nějaké ty znaky navíc. To se týká jak samotného stylu psaní (vkládání mezer a závorek, odřádkovávání apod.), tak také vhodného pojmenovávání datových typů, proměnných, funkcí a maker. Složitější výrazy je lepší zapisovat tak, aby byl jejich smysl snadno pochopitelný (tedy klidně i na víc řádků). Detaily o doporučeném formátování kódu v jádře naleznete v souboru `Documentation/CodingStyle`; i pokud vám tento konkrétní styl stoprocentně nevyhovuje, je dobré se ho držet, zejména existuje-li nějaká šance, že váš kód někdy zveřejníte, či jej dokonce budete chtít začlenit do hlavní větve jádra.
3. **Na dobrou přenositelnost hleděti budeš.** Kromě ovladačů zařízení, která jsou těsně spjata s určitou hardwarovou platformou (ale nejlépe ani tam), by se neměly nikde používat nepřenositelné postupy. Většinou to ani není nutné, existují totiž makra,

která se na příslušné platformě rozvinou přesně do té podoby, která by se použila při nepřenositelném přístupu – rychlost tedy netrpí.

4. **Nové systémové volání nevytvoříš.** Vytvoření nového systémového volání musí být výsledkem zralých úvah a širokého konsensu mezi vývojáři. Zaplevelování tabulky systémových volání na nějaké úzce specializované záležitosti by bylo cestou do pekel. Přidání nového volání „na vlastní pěst" navíc nic dobrého nepřinese – jen problémy s použitím takového volání.
5. **Na nové systémové volání hráti si nebudeš.** Vyslyšení předchozí rady nezřídka vede na jiné podobné řešení – emulaci nového systémového volání, většinou přes volání `ioctl()`. To je určeno k ovládání zařízení, tvůrci ovladačů ho ale zneužívají ke kdečemu. Pokud existuje nějaká vhodnější cesta, jak vyřešit komunikaci s ovladačem, je vždy lepší ji využít.
6. **Co není tvoje, na pokoji necháš.** Každý modul v jádře by se měl starat pouze a jen o sebe a vše ostatní nechat na pokoji. Pokud je potřeba využít nějaké služby jiného modulu či nějaké jiné části jádra, musí se tak činit zásadně prostřednictvím k tomu určených funkcí.
7. **Pamětí pečlivě šetřiti budeš.** Operační paměť patří k velmi vzácným zdrojům, i když se dnes někomu může zdát opak. Jenže velká síla Linuxu je kromě jiného i v tom, že se spokojí i s počítači, které hardwarovými prostředky neoplývají. Platí proto zásada, že se v jádře alokuje jen ta paměť, která bude opravdu potřeba, a po použití se co nejdřív uvolní. Pokud je nutné si připravit paměť dopředu, je potřeba bezpodmínečně dodržet správný postup.
8. **Nikde zdržovati se nebudeš.** Další striktní zásada v jádře – čas je velmi drahý. Extrémně to platí při obsluze přerušení a ve všech případech, kdy je něco zamčeno. S tím také souvisí důležité pravidlo, které velí vykonat na dané úrovni časové kritičnosti jen to, co je nezbytně nutné – zbytek se přesune tam, kde už není tolik naspěch.
9. **Chyby vždy řádně hlásiti budeš.** Zatímco v uživatelském programu lze do jisté míry přistupovat k obsluze chyb vlažně, v jádře je to absolutně nepřípustné. Každá návratová hodnota funkce se musí otestovat a v případě chyby je nutné na ni adekvátně zareagovat. Test na chyby má vždy absolutní přednost před rychlostí.
10. **Vždy po sobě řádně uklidíš.** Veškeré přivlastněné prostředky se musí bezpodmínečně vrátit systému. Cokoli se alokovalo, musí se uvolnit. Pokud něco nebude hned následně potřeba, je lepší to uvolnit okamžitě po použití. Posledním okamžikem je uvolňování modulu – bylo by hrubou chybou spoléhat na automatický úklid, nedávno začleněný do jádra.
11. **Problémům se souběhem zabráníš.** S každým sdíleným prostředkem je potřeba nakládat tak, aby nenastaly problémy se souběhem (race conditions). Při výběru vhodné metody je dobré začít těmi, které nevyžadují explicitní synchronizaci (např. atomické proměnné, *RCU*). Pokud už se musí něco zamykat, pak vždy vhodným zámkem a na co nejkratší dobu. Pozor také na vzájemné zablokování (deadlock).
12. **Nikomu a ničemu věřiti nebudeš.** Z uživatelského prostoru, ze sítě, ze zařízení atd. mohou přijít všelijaká data. Zásadou je ničemu nevěřit, všechno kontrolovat. Zvlášť

velký pozor je potřeba dávat na velikosti bufferů, aby nedošlo k jejich přetečení, které je kromě jiného zneužitelné k útoku.

13. **Citlivá data uniknout nenecháš.** Zejména se to týká operační paměti. Paměť se může využívat všelijak, takže když se alokuje nějaký blok paměti, většinou obsahuje data po předchozím použití. Při neopatrném zacházení se tato data mohou dostat do nepovolaných rukou. Proto je potřeba používanou paměť raději předem nulovat (což už je dnes velmi jednoduché) a nikdy nekopírovat větší bloky dat, než je nutné.
14. **Procesor usurpovati si nebudeš.** Nelze spoléhat na to, že jádro bude zkompilováno jako preemptivní, a plánovač tedy odejme úloze procesor. Naopak – systém by měl spolehlivě fungovat i tehdy, když se preemptivní chování vypne (v některých případech je totiž zbytečné). Pokud by tedy něco mělo trvat dlouho, je potřeba se postarat o včasné explicitní odevzdávání procesoru.
15. **Co hotového jest, použiješ.** V jádře je implementována spousta funkcí (byť zdaleka ne tolik jako ve standardní knihovně), které lze téměř používat dle libosti. Nemá smysl snažit se chytračit – vlastní řešení může být za některých situací rychlejší, jindy ale pomalejší nebo jinak problematické. Existující funkce jsou většinou dlouhodobě ověřené na různých platformách a jejich použití ve většině případů přináší optimální kompromis mezi přenositelností a výkonem, při výtečné spolehlivosti.

Pravidel je tak akorát, aby se dala dobře zapamatovat. Ideální je, když přejdou přímo „do krve", pak už se na ně nemusí vůbec myslet.



## V čem psát?

Tato otázka může mít dva významy. Jednak může znamenat programovací jazyk, jednak editor nebo vývojové prostředí. Podíváme se na oba významy.

Ohledně jazyka je to naprosto jednoznačné – jazykem volby bude **C**. Konkrétně jde o GNU C, což je v podstatě ISO C90 s některými prvky z ISO C99 a dalšími specialitami navíc. Dá se říci, že lze používat přímo ISO C90, kompilátor by s tím neměl mít problémy.

Proč nejde používat něco jiného, třeba C++? Je to jednoduché. Zrovna C++ by používat šlo, ale není to dobrý nápad. Důvodů je víc. Jednak nám C++ neposkytuje takovou kontrolu nad chováním kódu jako C, byly by komplikace s řešením výjimek (bez nich přicházíme o značnou část výhod C++), chybí standardní knihovna a v neposlední řadě některé verze kompilátoru generují poměrně mizerný a nepříliš stabilní kód.

Existují určité projekty, které se zabývají použitím jiných jazyků (než C) v jádře, ale mezi vývojáři panuje vcelku široká shoda, že optimálním jazykem pro jádro operačního systému je právě C. Pouze některé nízkoúrovňové operace má smysl psát přímo v assembleru, ale to je zase jiná kapitola.

Přejděme k samotnému psaní kódu. Tady platí pravý opak – každý si může zcela svobodně zvolit, v čem bude psát. Úplně stejně vyhoví VIM, Emacs, Kate, jEdit a jiné editory, stejně jako mohutná vývojová prostředí, kam na GNU/Linuxu patří třeba Eclipse (s pluginem CDT), Anjuta nebo Sun Studio. Záleží jen na osobních preferencích, výhodné je samozřejmě mít něco, co zvýraňuje syntaxi, umí napovídat názvy funkcí, dokáže párovat závorky a vůbec dělat všelijakou tu „černou práci", která je s programováním spjata.

## Standard psaní kódu

Podobně jako každý větší projekt, na němž pracuje více lidí, také linuxové jádro potřebuje mít něco jako *standard psaní kódu* (*coding standard*). Každý, kdo se účastní vývoje, by ho měl v zájmu co nejlepší spolupráce dodržovat, přestože je mu proti srsti a pro vlastní projekty používá něco jiného. Vývojáři jádra takový standard vytvořili – plné znění najdete (pochopitelně v angličtině) v souboru `Documentation/CodingStyle`. Neuškodí ale připomenout si z něj některé body.

- **Odsazování.** Odsazovat by se mělo tabulátorem, ne mezerami. Pokud jsou někde potřeba víc než 3 úrovně odsazení, je něco špatně. Nedávejte na jeden řádek více příkazů nebo přiřazení. Nenechávejte prázdné znaky (tabulátory nebo mezery) na koncích řádků.
- **Dlouhé řádky a řetězce.** Délka řádku by neměla překročit 80 znaků (což je typická délka řádku na konzole). Co vychází delší, mělo by se rozdělit, pokud to jde. Platí to i pro dlouhé textové řetězce.
- **Složené závorky.** Bloky, které nejsou funkcemi (`if`, `while`, `for` apod.), by měly mít otevírací závorku na stejném řádku jako uvozující příkaz. Pokud blok pokračuje (`else`), bude klíčové slovo následovat na stejném řádku za uzavírací závorkou. Funkce mají mít otevírací závorku na samostatném řádku.
- **Mezery.** Za většinou klíčových slov (typicky `if`, `while`, `switch` atd.) by měla následovat mezera. Naopak za některá klíčová slova (`sizeof`, `typeof`, `__attribute__`) se mezera nevkládá. Mezera nepatří ani z vnitřní strany kulatých závorek. Hvězdička u ukazatelových typů patří k názvu proměnné, ne k názvu typu. Kolem binárních operátorů patří mezery (nikdy ale dovnitř), naopak mezi unárním operátorem a identifikátorem být mezera nemá.
- **Názvy identifikátorů.** Měly by být stručné, ale výstižné. Kromě maker se používají malá písmena, slova se oddělují podtržítkem. Globální proměnné a funkce je vhodné pojmenovávat tak, aby byl význam dobře zřejmý. Lokální proměnné mají mít krátké názvy (`i`, `tmp` apod.). Je zbytečné, aby název obsahoval označení datového typu.
- **Definice typů.** Neměly by se definovat typy pro struktury a ukazatele, a obecně ani tam, kde pro to nejsou dobré důvody. Vhodné situace pro definování nových typů jsou tyto:
  1. Neprůhledné struktury, které se používají výhradně skrze předávání ukazatele a u kterých nemá být vidět dovnitř.
  2. Celočíselné typy, pokud je k tomu nějaký vážný důvod.
  3. Speciální typ pro kontrolu nástrojem *sparse*.
  4. Typy odpovídající typům podle ISO C99.
  5. Typy pro bezpečné použití v uživatelském prostoru.
- **Funkce.** Každá funkce by měla dělat jedinou operaci – a dělat ji dobře. Délka kódu ve funkci by neměla být větší než 2 obrazovky (po 24 řádcích). U exportovaných funkcí by mělo makro pro export následovat hned za implementací funkce (bez prázdného řádku). Prototypy funkcí by měly vždy obsahovat kromě typů argumentů i jejich názvy.

- **Centralizované opouštění funkcí.** Funkce by měla být napsána tak, aby běh končil v jediném bodě, zvlášť pokud se musí provádět nějaký úklid. Pro tyto účely lze s úspěchem používat (jinak nepříliš doporučované) skoky na návěští (`goto`).
- **Komentáře.** Je to dobrá věc, ale nesmí se to s nimi přehánět. Je lepší napsat dobrý a dobře srozumitelný kód, než napsat špatný a pak ho dlouze vysvětlovat. Komentář by měl říkat, **co** kód dělá, a ne **jak** to dělá. Pro funkce API (tj. vše, co se používá zvenku) by se měly vždy používat strukturované komentáře pro generování dokumentace (nástrojem *kernel-doc*). Je žádoucí komentovat i proměnné, už proto je dobré deklarovat jen jednu na řádek.
- **Nepořádek v kódu.** Dělá ho každý, ale je to řešitelné. Mnohé editory (a zvlášť vývojová prostředí) poskytují možnost automatického zformátování kódu. Je dobré to využívat, usnadní se tak práce a zpřehlední kód.
- **Konfigurační soubory pro kompilaci jádra.** Mají trochu jiný formát – odsazuje se sice tabulátorem, ale pro odsazení nápovědy se používají 2 mezery. Všechny experimentální věci by měly být „obklíčeny" podmíněným blokem pro `CONFIG_EXPERIMENTAL`. V popisu by se také měla objevit příslušná poznámka (`EXPERIMENTAL`). Nebezpečné volby by měly být také příslušně označeny (`DANGEROUS`).
- **Datové struktury.** Všechny struktury, ke kterým lze přistupovat z různých úloh, musí být opatřeny *počitadlem referencí* (*reference counter*). Lze tak zajistit, aby se paměť uvolnila (poloautomaticky) v okamžiku, kdy se struktura přestane používat. Důležitým předpokladem samozřejmě je, aby se počitadlo používalo pro veškeré přístupy (získání a uvolnění objektu).
- **Makra a výčtové typy.** Pro konstanty apod. se používají makra s názvy velkými písmeny. Makra chovající se jako funkce je dobré nahradit inline funkcemi. Pro větší počet souvisejících konstant je lepší (místo maker) používat výčtové typy (`enum`). Je důležité se zásadně vyvarovat:
  1. maker, která řídí běh (např. obsahují `return`).
  2. maker pracujících s lokálními proměnnými (spoléhajících na jejich název).
  3. maker použitých jako *l-hodnoty.*
  4. zanedbávání ochrany priorit operátorů (tj. situací, kdy operátory v argumentu mohou změnit pořadí operací uvnitř makra).
- **Zprávy jádra.** Pozor na gramatiku, překlepy apod. Na konci zprávy by neměla být tečka. Čísla je zbytečné dávat do závorek.
- **Alokace paměti.** Při specifikaci požadované velikosti paměti je nejlepší používat způsob `sizeof(*p)`, kde `p` je ukazatel příslušného typu (hodí se to při případné změně typu). Přetypovávání vrácené hodnoty (typu `void*`) na jiný ukazatel je zbytečné.
- **Inline funkce.** Neměly by se nadužívat; funkce delší než 3 řádky by se neměla deklarovat jako `inline` (zrychlení není tak významné, aby vykoupilo následky způsobené vyšší spotřebou fyzické paměti).
- **Návratové hodnoty funkcí.** Je dobré pečlivě rozlišovat, kdy má funkce vracet pravdivostní hodnotu (jen u predikátových funkcí, např. testujících stav) a kdy 0 při úspěchu a záporné číslo při chybě (většina ostatních funkcí). U exportovaných funkcí je to povin-

nost. Funkce vracející ukazatel by měly při chybě vracet `NULL` nebo chybovou hodnotu zakódovanou pomocí `ERR_PTR`.

- **Makra v hlavičkových souborech.** Není dobré definovat makra pro úkoly, pro které jsou již k dispozici – hlavičkové soubory jádra jich obsahují velké množství.
- **Modifikátor** `volatile`**.** Až na naprosté výjimky nemá v kódu jádra co pohledávat. Jeho cílem je zabránit kompilátoru v optimalizaci přístupu k proměnné, která se může asynchronně externě změnit. Rozhodně nemá sloužit jako „ochrana sdílených dat", protože tuto funkci stejně nemůže spolehlivě plnit.

Šedivá je teorie, ale realita je mnohem pestřejší. Některé popsané zásady se v jádře velmi zhusta nedodržují – zejména ty, které souvisejí s formátováním kódu. Můžete si toho všimnout v mnoha souborech z jádra hlavního stromu. Je to dáno kromě jiného i tím, že je některý kód převzat odjinud a nebyl formálně přepracován.

**Poznámka:** V některých případech je porušení zásad úmyslné. Např. u některých struktur pro ovladače zařízení USB najdeme názvy položek, které obsahují malá i velká písmena a navíc kódují i datový typ. Má to logiku – názvy odpovídají specifikaci USB. V tomto případě má přednost jednoznačná srozumitelnost před striktním dodržováním pravidel. Podobné „nesrovnalosti" najdeme například také u správy paměti.

# Jak kompilovat

Jádra řady 2.6 mají oproti těm dřívějším velkou výhodu. Kompilační systém *KBUILD* je připraven tak, aby kompilace celého jádra nebo třeba jen jediného modulu byla hračkou. Většinou nepotřebujeme kompilovat celé jádro – stačí těch pár souborů, které patří k našemu modulu. A k tomu si stačí jen tři věci:

- zdrojové soubory jádra,
- kompilátor GCC,
- soubor `Makefile`.

Zdrojové soubory pro oficiální jádro (*vanilla*) lze stáhnout ze serveru `kernel.org`, pro distribuční jádra pak ze serverů jednotlivých distributorů, případně jsou přiloženy na instalačním médiu. Kromě plných kódů lze použít i speciální balíky (označené např. `kernel-devel`), které obsahují KBUILD, hlavičkové soubory a některé další věci, ale neobsahují vlastní zdrojové kódy jádra. (Detaily o tom, jak získat aktuální zdrojové kódy jádra viz čtvrtou část knihy.)

Kompilátor je docela ošemetnou věcí. Není to tak dávno, co se GCC ve verzích 4.x vůbec nedoporučovaly pro kompilaci jádra (resp. s nimi ani jádro často nešlo zkompilovat), dnes už je situace lepší. Jedna důležitá zásada ale platí: modul do jádra by se měl kompilovat stejnou verzí, jakou bylo kompilováno jádro. Dodržení tohoto pravidla umožňuje předejít některým velmi záludným problémům vyplývajícím z toho, že se mezi verzemi může změnit generovaný kód.

Poslední potřebnou věcí je `Makefile` čili soubor pro řízení kompilačního procesu. Většinou bude velmi jednoduchý, půjde pouze o to, aby byly určeny správné zdrojové kódy jádra (odpovídající verze). K přesnému obsahu se dostaneme později.

**Tip:** Vlastní kompilační proces můžeme s výhodou spouštět přímo z vývojového prostředí. Pak se kompilace jádra prakticky nijak neliší od kompilace uživatelského programu.

## Zavádění zkompilovaného modulu

Po kompilaci můžeme zkusit modul zavést do jádra. Ovšem pozor – tato operace už je nebezpečná, protože může narušit fungování systému nebo způsobit úplné zatuhnutí. Ideální je, když se modul zkouší na samostatném stroji, na kterém jinak nic neběží, nejsou tam žádná důležitá data atd. Musí mít ovšem stejné jádro, pro jaké se kompilovalo.

Výbornou cestou je použití virtuálních strojů – například tak, že na jednom fyzickém stroji máme dva virtuální. Na jednom se kompiluje, na druhém testuje zkompilovaný modul. Případné poškození jednoho virtuálního systému nijak nenaruší druhý virtuální (a pochopitelně ani fyzický) systém. Výhoda tohoto uspořádání je i v tom, že lze zkoušet i různá jádra, bez ohledu na to, jaké je na fyzickém systému. Mnohdy to ale nelze použít, zejména pokud potřebujeme přístup k fyzickému zařízení.

Riziko následků na systému fyzického stroje lze minimalizovat jednak použitím vhodného souborového systému (se žurnálováním – např. ext3), a dále pak tím, že před načtením modulu spustíme příkaz `sync`. Tím se uloží na disk veškerá data čekající v paměti. Jinak by se mohlo stát, že by byla ztracena třeba i poslední změna ve zdrojovém souboru modulu. Pokud dojde k vážné chybě (viz kap. 4), musí se systém co nejdřív restartovat. Podobně i v případě, že při pokusu o uvolnění modulu toto selže, typicky kvůli nějakému zapomenutému prostředku.

## Datové typy

Problematika datových typů je při vývoji pro jádro jednou z naprosto klíčových. Chyby v této oblasti bývají velmi ošemetné, špatně dohledatelné a mající dopady především na přenositelnost. Proto je při volbě datového typu pro uložení nějaké hodnoty dobré se na chvíli zamyslet a zvážit všechny aspekty.

V jádře můžeme používat následující datové typy:

- klasické celočíselné typy jazyka C,
- celočíselné typy s pevnou délkou,
- struktury,
- uniony,
- výčtové typy,
- pole,
- ukazatele,
- speciální typy,
- seznamy.

Zejména nelze v rámci jádra používat typy floating point (plovoucí řádová čárka). Jádro totiž musí být schopné běžet i na procesorech bez matematického koprocesoru; případné využití koprocesoru může mít navíc v některých kontextech nepředvídatelné následky.

## Celočíselné typy

Klasické celočíselné typy (`int`, `unsigned int`, `long int`, `unsigned short int` atd.) se používají velmi často. Hodí se pro vnitřní použití v jádře (bez komunikace s uživatelským prostorem) a obecně tam, kde buď nepotřebujeme pevnou délku, nebo naopak se má délka měnit podle architektury. Většinou vycházíme z toho, jaké typy přijímají a vracejí funkce API jádra a jaké typy jsou obsaženy v používaných datových strukturách.

**Tip:** Pro nejobecnější použití (obecné čítače a řídicí proměnné o malém rozsahu hodnot, návratové hodnoty výkonných funkcí, stavové informace apod.) je nejvhodnější typ `int`. Pracuje se s ním rychle (často dokonce atomicky), současně nezabírá zbytečně mnoho místa. Stejný typ konvenčně používáme i pro pravdivostní hodnoty.

Typ `char` má pevnou délku 8 bitů. Velikost ostatních celočíselných typů záleží na konkrétní architektuře; např. na i386 má `short` 16 bitů a `int` a `long` mají 32 bitů; gcc poskytuje i 64bitový typ `long long`.

Celočíselné typy s pevnou délkou (`s8`, `u16` apod.) se hodí tam, kde pracujeme s přesně danou velikostí hodnoty – např. při komunikaci se zařízením, při čtení datových formátů apod. Hodí se také pro komunikaci s uživatelským prostorem, protože nemusíme řešit přenositelnost mezi architekturami (`u32` prostě bude mít 32 bitů na kterékoli platformě).

**Tip:** Ve všech deklaracích, které se budou používat i v uživatelském prostoru, používejte místo jednoduchého `u16`, `s64` apod. podtržítkové identifikátory, tedy například `__u16` a `__s64`. Je to potřeba proto, že některé programy a knihovny definují stejně nazvané typy, často pomocí odlišných základních typů. Tímto se vyhneme riziku nepříjemných kolizí.

## Struktury

Struktury se v jádře používají velmi často. Nezřídka se s jejich pomocí emuluje objektovost programu, protože obsahují kromě datových položek také ukazatele na funkce. Obecně se strukturami není žádný problém – pozor ovšem na správnou inicializaci. Je dobré vždy po alokaci strukturu vynulovat (pokud nebyla alokována funkcí, která to udělá sama), a pokud existuje inicializační funkce, zavolat ji (případně předem nastavit potřebné hodnoty).

Jak si můžete pamatovat z pravidel pro psaní kódu, struktury často obsahují počitadlo referencí. Používá se obvykle tam, kde se ke struktuře přistupuje z více míst, aby bylo zajištěno bezpečné smazání v okamžiku, kdy data už nikdo nepoužívá. Pozor na to, že jakýkoli přístup k datům ve struktuře znamená, že si ji musíme nejdřív odněkud vyžádat a bezpodmínečně zkontrolovat, zda to bylo úspěšné. Může se totiž stát, že se mezitím struktura zruší, a přístupová funkce tedy vrátí `NULL`. Úspěšné získání inkrementuje počitadlo referencí. Po použití strukturu uvolníme, čímž se dekrementuje stav počitadla (a pokud dosáhl 0, stuktura se zruší).

Jádro obsahuje strukturu s počitadlem referencí určenou k obecnému použití. Je to struktura kobject a podrobné informace o ní najdete v kapitole 13.

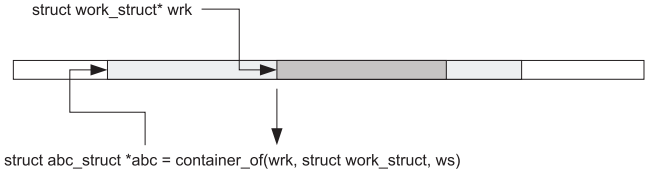
Se strukturami souvisí jeden zajímavý a velmi často používaný hack. Používá se zvlášť u vnořených struktur, ale i jinde. Je to makro `container_of()`, které umožňuje snadno zjistit adresu vnější struktury podle adresy prvku v této struktuře. Vezměme si následující příklad:

```
struct abc_struct {
  struct work_struct ws;
  ...
};
```

Ve funkci získáme ukazatel na položku `ws`, ale přitom potřebujeme ukazatel na celou strukturu. Stačí udělat toto:

```
struct work_struct *wrk = ...
struct abc_struct *abc = container_of(wrk, struct work_struct, ws);
```

Jednoduché, že? V hlavičkových souborech jádra je mnoho podobných maker, většinou ale vycházejí ze základního `container_of()`. Stojí za to vědět ještě alespoň o makru `offsetof()`, které vrátí offset dané položky ve struktuře.




**Obrázek 3.1** Makro container_of()

## Uniony a výčtové typy

Uniony nepatří k příliš rozšířeným typům, ale stejně je občas potkáme – například u struktur pro obsluhu signálů, hlaviček paketů, mechanismu *epoll* atd. Je potřeba si dávat pozor v případě, že střídáme různě dlouhé typy. Může se například stát, že jeden typ v unionu vynulujeme a jiný (delší) ještě obsahuje nějaká další data, která nebudeme očekávat. Tedy pokud se union nuluje, musí se vždy vynulovat celý.

Výčtové typy jsou o něco častější, přestože mnoho „kandidátů" je definováno pomocí maker. Standard doporučuje preferovat výčtové typy před makry. Je opravdu dobré je používat – už proto, že není potřeba se zabývat skutečnými hodnotami tam, kde si vystačíme se symboly. Pozor ovšem na to, že pokud výčtový typ neobsahuje žádnou zápornou hodnotu, bude bezznaménkový (typicky jako `unsigned int`), kdežto se zápornými hodnotami bude znaménkový.

## Pole a ukazatele

Velmi často používaným typem je pole, nejčastěji pole znaků. Práce s polem se nijak znatelně neliší od jeho použití v běžných programech. Hlavní rozdíl spočívá v tom, že nelze jen tak alokovat lokální pole na zásobníku. Ten je totiž velmi malý (obvykle 4 KB) a už jen trochu větší pole by přeteklo mimo. Proto musíme pole alokovat buď staticky (bude tedy po celou dobu přítomnosti modulu v jádře zabírat paměť), nebo dynamicky (a bude se muset explicitně uvolnit).

> **Poznámka:** Hlídání mezí polí je v jádře mnohem důležitější než v běžném programu. Protože přístup do paměti není omezen, hrozí přepsání paměti patřící kterékoli části jádra nebo některému z procesů.

Jednou z typických věcí v jádře je intenzivní používání ukazatelů – mnohem častější než v běžných programech. Je to proto, že velmi často používáme explicitně alokovanou paměť. Navíc se kvůli rychlosti nepoužívá volání hodnotou, aby se zbytečně nekopírovaly větší datové typy.

V kapitole o práci s pamětí se podíváme detailně, jak se pracuje s jednotlivými druhy paměti. Již nyní ale může být užitečné trochu předběhnout a říci si, že běžné ukazatele se používají při práci s virtuální pamětí jádra. Pracuje se s nimi prakticky stejně jako s ukazateli v uživatelském programu – lze je vzájemně přetypovávat (včetně implicitního přetypování), provádět dereferenci atd.

Zvláštním případem jsou ukazatele do paměti procesu (do uživatelského prostoru). Ačkoliv jsou to normální ukazatele (získané přes systémové volání), nesmíme je v žádném případě používat jinak než prostřednictvím k tomu určených funkcí – a to ani v případě, že by to při zkoušce prošlo (např. vyzkoušením prosté dereference). Na jiné architektuře nebo s jinou konfigurací to fungovat nebude. Navíc v případě, že příslušná paměťová stránka není k dispozici (např. byla odstránkována), to dopadne tragicky.

**Tip:** Ukazatele do uživatelského prostoru poznáme podle symbolu __user (pozor – jeho nepřítomnost nezaručuje, že nejde o takový ukazatel). Kompilátor symbol sice ignoruje, ale některé nástroje (např. sparse) umožňují ověřit, zda se s takovým ukazatelem neděje něco nepřístojného.

## Speciální typy

Jádro, podobně jako většina knihoven a některé uživatelské programy, definuje a používá mnoho různých speciálních typů. Často jsou to jen aliasy k celočíselným typům. Někdy jde o struktury, u kterých se běžně nesahá na datové položky. Většina deklarovaných struktur ale není definována jako typ – musíme tedy používat klíčové slovo `struct`. Ve výčtu speciálních typů nelze zapomenout ani na funkční typy.

Některé takové typy známe z běžných programů – např. `size_t`, `loff_t`, `time_t`, `uid_t` a mnohé další. Jádro si přidefinovává ještě mnoho jiných, kupříkladu `wait_queue_t`, `ctl_table`, `atomic_t` a různé jiné. Dále máme mnoho struktur obecných (`timespec`, `stat`, `pollfd`...) i speciálních (`task_struct`, `elevator_type`, `file_operations` atd.).

Používání typů má svůj význam. Pokud se jedná o datovou strukturu, je to jasné. Ale mnohdy se to týká i ostatních typů. Jednak se tím zlepšuje přenositelnost (typy mají přesně ty vlastnosti – např. pevnou délku, nebo naopak odlišnou podle platformy), a současně lze snadno kontrolovat, zda se někde neděje něco špatného.

**Poznámka:** Typickým případem je sběrnicová adresa paměti DMA. Ta má totiž význam jen pro zařízení, i když na některých platformách může být shodná s běžnou fyzickou adresou. S touto adresou by se mělo vždy pracovat jen prostřednictvím určených funkcí. Využití speciálního datového typu umožňuje odhalit „nelegální" použití takové adresy.

Zajímavým případem speciálního typu je pravdivostní typ `bool`. Byl zaveden teprve nedávno (konkrétně od verze 2.6.19), proto se zatím v jádře vyskytuje poměrně málo. U nového kódu by ale měl mít přednost před dosud používaným typem `int`."

## Seznamy

V jádře se velmi často pracuje s obousměrně zřetězenými spojovými seznamy. Proto jádro obsahuje přímo podporu pro takové seznamy, a tak není potřeba si je vytvářet pro každý ovladač samostatně. Hlavičkový soubor `linux/list.h` obsahuje vše potřebné pro základní seznamy, na dalších místech jsou i nějaké rozšiřující věci.

Seznam je založen na prvcích tvořených strukturami `list_head`. Struktura má dva prvky, a to ukazatele na předchozí a následující prvek. Samostatný prvek se inicializuje (staticky nebo dynamicky), po inicializaci oba jeho prvky obsahují ukazatele na sebe sama.

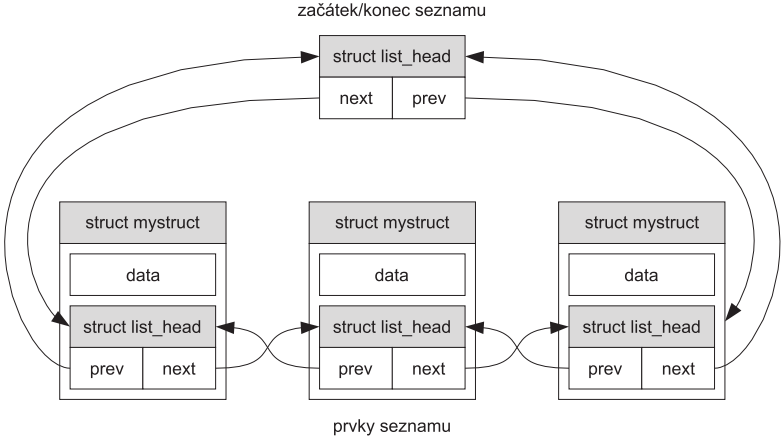
Pro vlastní práci se seznamem samozřejmě potřebujeme ještě začátek a konec. Můžeme mít buď proměnné s ukazateli na strukturu list_head, nebo přímo tuto strukturu, odpovídajícím způsobem inicializovanou. S prvky v seznamu manipulujeme pomocí funkcí `list_add()`, `list_add_tail()` a `list_del()`. První přidá prvek před určený prvek, druhá naopak za prvek a třetí odstraní prvek ze seznamu. Tady je krátký příklad:

```
struct list_head list;
LIST_HEAD_INIT(&list);

struct list_head *le = kzalloc(sizeof(*le), GFP_KERNEL);
LIST_HEAD_INIT(le);
list_add(le, &list);
```

V příkladu se nejdřív připraví samotný seznam, pak se alokuje paměť pro nový prvek (viz kap. 6), ten se inicializuje a pak přidá do seznamu. Kromě uvedených operací máme například také `list_replace()` pro náhradu prvku, `list_move()` a `list_move_tail()` pro jeho přesun a různé další funkce.

Logickou otázkou je, jak do tohoto seznamu vkládat nějaké větší prvky, například instance zařízení. Odpověď se nachází o kousek zpátky, u datových struktur. Ano, jsou to vnořené struktury a makro `container_of()`. Ve funkcích pro práce se seznamem se normálně používají ukazatele na vnořené struktury `list_head` a v případě potřeby se z nich získávají ukazatele na nadřazené struktury. Prosté a jednoduché.




**Obrázek 3.2** Využití struktury list_head pro tvorbu seznamů

**Tip:** Místo makra `container_of()` lze používat též `list_entry()`. Je to úplně totéž, jen s jiným názvem. K dispozici je též řada maker pro hromadné (iterační) operace prováděné nad celým seznamem.

# Další doporučení

Při vývoji součástí pro jádro lze s úspěchem využít některé vlastnosti a předem připravené věci. Lze tak zrychlit vývoj a zkvalitnit výsledné dílo.

## Optimalizační makra

Při vyhodnocování podmínky kompilátor běžně nemá informace o tom, jak pravděpodobná je která větev. Proto se může stát, že se kód uspořádá tak, že to bude z hlediska rychlosti nevýhodné – bude se často chodit delší cestou, kdežto cesta kratší se použije v málo případech. Tomu lze předejít použitím optimalizačních maker `likely()` a `unlikely()`.

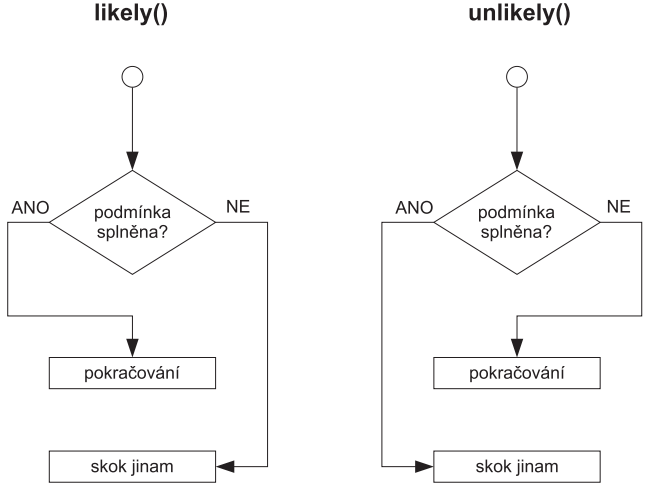
Makra může vývojář použít jako informaci pro kompilátor, jak má kód nejlépe seřadit. Podívejme se na následující příklad:

```
void *ptr = kmalloc(100, GFP_KERNEL);
if (ptr == NULL)
  return -ENOMEM;
```

Kód v příkladu se pokusí alokovat paměť, a pokud to selže, vyskočí z funkce s vrácením chyby. Pokud to bude napsáno takto, kompilátor si může instrukce seřadit jakkoli. Protože ale víme, že alokace bude téměř vždy úspěšná (a selže jen naprosto výjimečně), můžeme to kompilátoru říct:

```
void *ptr = kmalloc(100, GFP_KERNEL);
if (unlikely(ptr == NULL))
  return -ENOMEM;
```

Takto napsaný kód se bude kompilátor snažit optimalizovat pro nesplnění podmínky – pokud tedy bude alokace úspěšná, půjde se kratší cestou. Pokud by se mělo optimalizovat naopak na splnění podmínky, použilo by se makro `likely()`.




**Obrázek 3.3** Optimalizační makra likely() a unlikely()

**Poznámka:** Kompilátor obecně nemusí brát tato makra v úvahu. Pak se ale nic nestane, bude to jako kdyby tam žádná makra nebyla.

## Mechanismy uložení dat

Podobně jako v případě seznamů i v dalších podobných případech se vyplatí nejdříve se podívat, zda už jádro nenabízí implementaci pro daný účel. Taková implementace bude pak společná pro celé jádro, čímž se výrazně zjednoduší a zrychlí implementace, ušetří paměť, navíc je kód dobře otestovaný a odladěný.

Jednotlivé implementace se nacházejí v adresáři `lib` hlavního adresáře jádra. To ale není tak podstatné, vývojáře zajímají spíš hlavičkové soubory – a ty jsou v adresáři `include/linux`. Jádro obsahuje následující datové mechanismy:

- bitová mapa (`bitmap.h`),
- prioritní vyhledávací strom (`prio_tree.h`),
- radixový strom (`radix-tree.h`),
- červeno-černý strom (`rbtree.h`),
- prioritní seznam (`plist.h`).

Je docela možné, že se v pozdějších verzích objeví nějaké další takové mechanismy.

## Algoritmy v jádře

Co bylo řečeno o uložení dat, týká se také algoritmů pro manipulaci s daty. I tady samozřejmě platí, že je zbytečné znovu implementovat něco, co už v jádře je. Algoritmy lze rozdělit do několika skupin:

- kontrolní součty a hashe (CRC16, CRC32, MD5, SHA1, Tiger apod.),
- šifrovací algoritmy (AES, BlowFish, FCrypt atd.),
- datová komprese (deflate),
- pseudonáhodná čísla (random32),
- řazení dat (heapsort),
- vyhledávání v textu (konečný automat, Boyer-Moore, Knuth-Morris-Pratt).

Některé z algoritmů (jako třeba zmíněné vyhledávání v textu) jsou dokonce modulární – lze si doimplementovat speciální modul pro určité účely a používat ho, je-li k dispozici.