VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EFEKTIVNÍ SPRÁVA PAMĚTI VE VÍCEVLÁKNOVÝCH APLIKACÍCH

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. LIBOR VAŠÍČEK
AUTHOR

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EFEKTIVNÍ SPRÁVA PAMĚTI VE VÍCEVLÁKNOVÝCH APLIKACÍCH

EFFECTIVE MEMORY MANAGEMENT FOR MULTI-THREADED APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. LIBOR VAŠÍČEK
AUTHOR

VEDOUCÍ PRÁCE Ing. KAREL MASAŘÍK
SUPERVISOR

BRNO 2008

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2007/

Zadání diplomové práce

Řešitel: **Vašíček Libor, Bc.**
Obor: Informační systémy
Téma: **Efektivní správa paměti ve vícevláknových aplikacích**
Kategorie: Operační systémy

Pokyny:

1. Prostudujte možnosti virtualní paměti operačních systemů Windows/Linux.
2. Seznamte se s algoritmy First fit, Buddy system a Slab alokátor.
3. Navrhněte metodu pro efektivní alokaci paměti ve vícevláknovém prostředí. Zvolte vhodné synchronizační nástroje a uvažujte o možnosti využití vyrovnávacích pamětí, které sníží používání synchronizace. Do návrhu začleňte nástroje pro ladění a měření výkonu aplikací.
4. Danou metodu implementujte formou statické knihovny pro Windows/Linux.
5. Proveďte srovnávací testy se standardnímy alokátory.

Literatura:

- Daniel P. Bovet, Marco Cesati: Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005, ISBN 0-596-00565-2
- Mel Gorman: Understanding the Linux Virtual Memory Manager, Prentice Hall, 2004, ISBN 0-13-145348-3
- Mark E. Russinovich, David A. Solomon: Microsoft Windows Internals, Fourth Edition, Microsoft Press, 2004, ISBN 0735619174
- Dále dle pokynů vedoucího

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění prvních 3 bodů zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVR-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Masařík Karel, Ing.**, UIFS FIT VUT
Datum zadání: 24. září 2007
Datum odevzdání: 19. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Libor Vašíček**
Id studenta: 89994
Bytem: Čelakovského 35/1716, 678 01 Blansko
Narozen: 12. 03. 1981, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   diplomová práce

Název VŠKP: Efektivní správa paměti ve vícevláknových aplikacích
Vedoucí/školitel VŠKP: Masařík Karel, Ing.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☒ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

Tato práce popisuje návrh a implementaci efektivní správy paměti pro vícevláknové aplikace. Nejprve jsou popsány možnosti virtuální paměti, jež poskytují moderní operační systémy, mezi něž patří Microsoft Windows a Linux. Dále jsou vysvětleny nejpoužívanější algoritmy správy paměti. Jejich vlastnosti jsou následně vhodně použity při návrhu nového správce paměti. Výsledný návrh obsahuje určité nástroje pro ladění a měření výkonu aplikací. V závěru práce byla provedena série testů a zhodnocení dosažených výsledků.

Klíčová slova

Virtuální paměť, stránkování, segmentace, výpadek stránky, ochrana paměti, manuální správa paměti, fragmentace paměti, alokace paměti, sequential fit, boundary tag, first fit, best fit, buddy systém, slab alokátor, vlákno, vyrovnávací paměť, únik paměti, přetečení bufferu.

Abstract

This thesis describes design and implementation of effective memory management for multi-threaded applications. At first, the virtual memory possibilities are described, which can be found in the latest operating systems, such as Microsoft Windows and Linux. Afterwards the most frequently used algorithms for memory management are explained. Consequently, their features are used properly for a new memory manager. Final design includes particular tools for application debugging and profiling. At the end of the thesis a series of tests and evaluation of achieved results were done.

Keywords

Virtual memory, paging, segmentation, page fault, memory protection, manual memory management, fragmentation of memory, allocation, sequential fit, boundary tag, first fit, best fit, buddy system, slab allocator, thread, cache, memory leak, buffer overflow.

Citace

Libor Vašíček: Efektivní správa paměti ve vícevláknových aplikacích, diplomová práce, Brno, FIT VUT v Brně, 2008

Efektivní správa paměti ve vícevláknových aplikacích

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Karla Masaříka. Další informace mi poskytla firma ZONER software, s.r.o. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Libor Vašíček
19. května 2008

Poděkování

Především bych rád poděkoval mému vedoucímu diplomové práce Ing. Karelu Masaříkovi, který mi umožnil pracovat na tomto diplomovém projektu a věnoval mi čas při konzultacích. Také bych chtěl poděkovat firmě ZONER software, s.r.o., která byla externím zadavatelem tohoto projektu a poskytla mi technické zázemí a cenné informace.

© Libor Vašíček, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 3
2 Virtuální paměť 5
2.1 Stránkování 6
2.1.1 Víceúrovňové stránkování 7
2.1.2 Segmentace 9
2.1.3 Výpadek stránky 11
2.1.4 Zavádění stránek na žádost 13
2.1.5 Ochrana paměti 13
2.2 Prostředky správy paměti poskytované OS 13
3 Paměťová hierarchie 14
3.1 Vyrovnávací paměť (cache) 15
3.1.1 Plně asociativní 15
3.1.2 Přímo mapovaná 16
3.1.3 S omezeným stupněm asociativity 17
3.1.4 Barvení cache 18
4 Synchronizace 19
4.1 Synchronizační nástroje 19
4.1.1 Spinlock 20
5 Manuální správa paměti 25
5.1 Fragmentace paměti 25
5.1.1 Interní fragmentace 26
5.1.2 Externí fragmentace 26
5.1.3 Fragmentace při stránkování a segmentaci 26
5.2 Alokátory s bitovou mapou 27
5.3 Rodina alokátorů „Sequential Fit“ 28
5.3.1 First fit 29
5.3.2 Next fit 29
5.3.3 Best fit 30
5.4 Alokátory s oddělenými seznamy 30
5.4.1 Přidělování s omezenou velikostí bloku (Buddy system) 31
5.4.2 SLAB alokátor 33
5.5 Urychlení alokace s využitím cache 35
5.6 Optimální alokátor 36

6 Koncept a implementace efektivního alokátoru 39
6.1 Centrální správa paměti 39
6.1.1 Navracení systémové paměti 40
6.2 Alokátor velkých objektů 41
6.2.1 Logické schéma alokátoru 41
6.2.2 Způsob přidělení a uvolnění paměti 43
6.2.3 Vyhledání volného úseku 43
6.2.4 Deskriptory úseků paměti 44
6.3 Alokátor malých objektů (Slab alokátor) 45
6.3.1 Buddy systém 45
6.3.2 Vytvoření Slab cache 46
6.3.3 Přidělení a uvolnění paměti 47
6.3.4 Použití vyrovnávacích pamětí 48
6.4 Implementace funkce malloc 49
6.4.1 Inicializace a uvolnění alokátorů 50
6.4.2 Vytvoření vektoru velikostí Slab cache 51
6.4.3 Nahrazení standardní funkce malloc 51
7 Nástroje pro kontrolu paměti 52
7.1 Implementace kontroly paměti 52
7.1.1 Překročení hranice paměti 52
7.1.2 Nesprávné uvolňování paměti 55
7.1.3 Únik paměti (memory leak) 56
7.1.4 Neošetření návratové hodnoty NULL 58
7.2 Statistiky alokátoru 60
8 Srovnávací testy 61
8.1 Synchronizace 61
8.2 Porovnání správců paměti 63
8.2.1 Standardní správce systému Linux 63
8.2.2 Standardní správce systému Windows 66
Závěr 70
Literatura 71
Seznam příloh 73
Příloha A – Správa paměti OS Windows 74
Příloha B – Správa paměti OS Linux 75

1 Úvod

Nejspíše každý z nás, kdo začínal programovat, zjistil, že existují datové struktury, které nemají předem známou velikost. V takových případech již nestačilo statické přidělování paměti, které implicitně poskytuje každý programovací jazyk. Je možné, že se našlo pár bystrých programátorů, kterým se nechtělo hledat ve specifikaci jejich programovacího jazyka a napadlo je staticky rezervovat úsek paměti, ze kterého postupně přidělovali menší bloky o dané velikosti. Možná to může někomu na první pohled připadat směšné, ale počítače do 70. let minulého století disponovaly tak malou pamětí, že o správě dynamické paměti se v té době ani nedalo mluvit a aplikace byly psány právě tímto způsobem. Takový příklad můžeme najít stále i v dnešní době, kdy máme vestavěné zařízení téměř na každém kroku a mnohdy na nich neběží žádný operační systém, který by to umožnil. Pokud se ale vyhneme těmto případům, tak nám většina současných jazyků nějakou formou zpřístupňuje správu paměti, která je většinou dodávána jako knihovní funkce nebo je přímo součásti daného jazyka.

V případě, že se podíváme na nějakou definici pojmu „správa paměti“, tak zjistíme, že se jedná v podstatě o organizaci a hospodaření s operační pamětí. V nejjednodušší formě zahrnuje pouze přidělování paměti. Většinou je taková funkčnost nedostačující a proto zajišťuje i následné uvolnění paměti (někdy též nazývané regenerace paměti). Způsoby jakým je uvolňování dosaženo, rozdělují správu paměti na manuální a automatickou. Jak již název napovídá, tak při manuální správě musíme sami volat uvolňování paměti, kdežto automatická sama zajistí uvolnění nepotřebné paměti. Automatickou správou (zvanou též garbage collector, „sběrač odpadu“) se v této práci nebudu zabývat. Lze ji nalézt u většiny abstraktních objektových jazyků, které jsou interpretované, a tuto správu zajišťuje běhový virtuální stroj. U nízko-úrovňových jazyků, jako je např. C/C++, se s tímto způsobem správy paměti lze setkat maximálně ve formě speciálních knihoven. Nicméně od těchto jazyků se především vyžaduje efektivita, a proto se standardně používá manuální správa, která na druhou stranu může přinášet nepříjemné chyby, tak často spojované právě s jazykem C.

U moderních počítačových systémů hovoříme o úrovních správy paměti. Tyto úrovně jsou obvykle tři a vzájemně se doplňují: úroveň technického vybavení, operačního systému a aplikačních programů. V této práci se zaměřím na správu paměti na aplikační vrstvě. Nicméně, aby bylo přidělování paměti opravdu efektivní, musíme se dotknout všech vrstev a přesně pochopit jejich funkci. Slovem efektivní se myslí přidělení paměti o dané velikosti v co nejkratším čase, což je jedna z hlavních metrik daného algoritmu.

S rozvojem víceuživatelských a víceprocesorových systémů je čím dál větší potřeba efektivně sdílet paměťové prostředky. Počet procesorů a velikost paměti u jedné stanice neustále roste a špatná alokace paměti může velmi negativně ovlivnit celkový výkon aplikací, které alokují za jednotku času velké množství paměťových objektů. Typickým příkladem může být zatížený webový server.

Nicméně, tomuto problému se již nevyhnou ani domácí počítače, které často obsahují dvou-jádrové procesory. Navíc většina aplikací je dnes psaná v jazyce C++, který často vytváří objekty pomocí operátorů new, což je vlastně zabalené volání správce paměti.

Myslel jsem, že výrobci kompilátorů jazyka C/C++ na tento fakt mysleli a již za nás vytvořili velice efektivní správce paměti v dodávané standardní knihovně. Bohužel u systému Windows tomu tak není. Především tato skutečnost mě přiměla začít se zabývat tímto problémem a pokusit se vytvořit lepší správu paměti pro více-vláknové aplikace na uživatelské úrovni ve formě statické knihovny. Protože je jazyk C/C++ standardizovaný, rozhodl jsem se knihovnu vytvořit pro systém Windows i Linux současně.

2 Virtuální paměť

Virtuální paměť [1] nebo také adresování virtuální paměti je abstrakce implementovaná v počítačových systémech, která má za cíl navodit spuštěné aplikaci iluzi souvislé paměti, kterou může bez omezení používat. Ve skutečnosti tato paměť může být fragmentovaná nebo dokonce odložena na pevném disku. Díky tomuto mechanismu lze poskytnout virtuální paměť významně větší než je skutečná velikost fyzické paměti.

Všechny moderní operační systémy, mezi něž se nepochybně řadí Microsoft Windows a Linux, poskytují systém virtuální paměti [2]. Ve starších systémech, jako byl DOS z 80. let nebo u některých OS sálových počítačů z 60. let, tento mechanismus nebyl implementován. Princip virtuální paměti byl vyvíjen přibližně v letech 1959-1962 na „University of Manchester for the Atlas Computer“. Koncem 60. let se začal stávat standardem OS sálových počítačů. Zprvu byla virtuální paměť implementována pomocí segmentace paměti, později bylo vyvinuto stránkování, které se dnes nejvíce používá, a jeho princip si popíšeme v následující kapitole. Některé procesory (Intel) využívají zároveň se stránkováním i segmentaci.

Virtuální paměť je technicky implementována v hardwaru i softwaru což umožňuje mapování nesouvislé (fragmentované) fyzické paměti na logickou souvislou paměť. Všechny současné implementace umožňují operačnímu systému poskytnout následující vlastnosti:

- Paměťové místo může být adresováno, i když není aktuálně ve fyzické paměti. Hardware za přispění operačního systému automaticky nahraje požadovaná data z pomocného úložného prostoru tak, aby aplikace požadující danou adresu vůbec nic nepostřehla. Toto umožní programům přirozeně se odkazovat na více paměti, než skutečně v počítači existuje.
- Multiprogramové operační systémy mohou dokonce poskytnout více virtuálních adresových prostorů, které jsou navzájem zcela izolovány. Každá úloha může mít díky tomu vlastní adresový prostor, a tak může být přirozeně izolována od ostatních úloh ovlivňujících obsah paměti. Takto lze zvýšit odolnost proti chybám vzniklým v jiné úloze a celkovou robustnost operačního systému.
- Adresové prostory lze navzájem sdílet. Tento mechanismus umožňuje implementovat mezi-procesovou komunikaci („Inter-Process Communication“ – IPC), která je nezbytná pro předávání zpráv, synchronizaci nebo prosté sdílení paměti.

Zvýšené nároky na hardwarové a systémové prostředky jsou zanedbatelné v porovnání s výhodami, které používání virtuální paměti přináší. Je to menší potřeba fyzické paměti, jejíž význam se snižováním cen pamětí ale neustále klesá. Mnohem významnější je přínos v podobě efektivnějšího zavádění procesů do paměti. Díky odděleným adresovým prostorům může operační systém nahrát

program na fixní místo, aniž by ovlivnil ostatní procesy, a tak minimalizovat relokaci odkazů, která by jinak byla nutná. Z dalších výhod jmenujme menší spotřebu paměti, kterou umožňuje sdílení společného kódu pomocí dynamických knihoven a mechanismus „Copy-on-Write“ (COW), který zkopíruje celou oblast sdílené paměti až při pokusu o její modifikaci. Vedlejším efektem odkládání adresového prostoru na pevný disk je mapování souborů do paměti. Díky tomu lze velice efektivně pracovat se soubory bez nutnosti explicitního kopírování souboru do/z paměti pomocí typických systémových IO funkcí read/write. Samozřejmě nelze opomenout ochranu paměti, která radikálně zvýšila bezpečnost a stabilitu současných operačních systémů. Tato ochrana byla docílena různými stupni přístupových práv k určité oblasti paměti, díky čemuž je možné oddělit adresový prostor jádra operačního systému a uživatelských procesů. Aby vše mohlo fungovat, musí v procesoru existovat privilegované instrukce, ke kterým má přístup pouze jádro systému. Další možnost ochrany paměti je nastavení pouze pro čtení, zápis nebo spouštění kódu.

Toto všechno by bez podpory na straně hardwaru nebylo možné vytvořit, a proto mnoho moderních procesorů obsahuje „on-chip memory management unit“ (MMU). Jednotka zajišťující překlad virtuální adresy na fyzickou se označuje zkratkou TLB („Translation Lookaside Buffer“) a podrobněji si ji popíšeme v jedné z následujících kapitol.

Na závěr tohoto úvodu si v definici 2.1 zavedeme mapovací funkci, která představuje matematický zápis principu virtuální paměti:

$\mathbf{V} = \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{n} - \mathbf{1}\}$	Virtuální adresový prostor velikosti $\boldsymbol{n}$.
$\mathbf{M} = \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{m} - \mathbf{1}\}$	Fyzický adresový prostor velikosti $\boldsymbol{m}$, kde $\boldsymbol{m} \ll \boldsymbol{n}$.
$\boldsymbol{MAP}: \boldsymbol{V} \rightarrow \boldsymbol{M} \cup \{\boldsymbol{O}\}$	Zobrazovací – mapovací funkce adres.

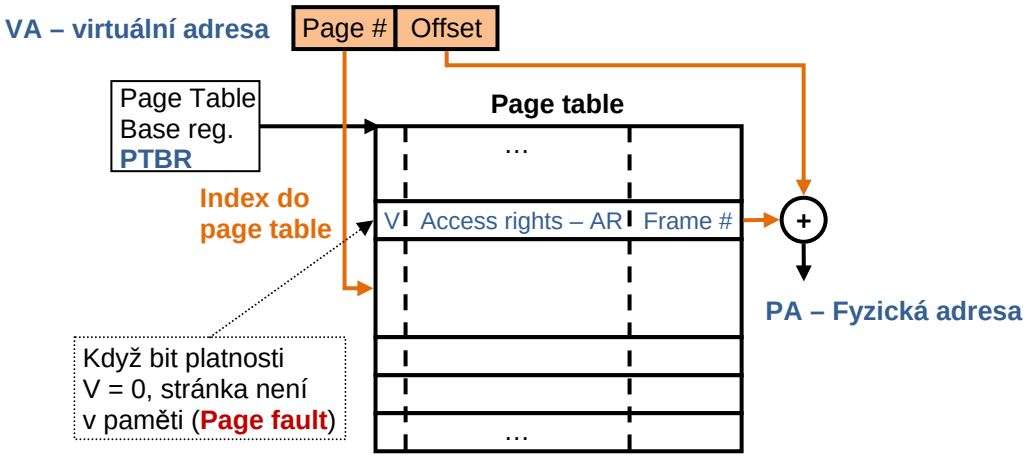
$$MAP(a) = \begin{cases} b\text{: Data na virtuální adrese } a \in V \text{ jsou reprezentována daty fyzické paměti } b \in M. \\ O\text{: Data na virtuální adrese } a \in V \text{ nejsou obsažena ve fyziké paměti.} \end{cases}$$

Definice 2.1: Mapování virtuálních adres na fyzické adresy [3]

2.1 Stránkování

Jeden z nejrozšířenějších způsobů implementace virtuální paměti je tzv. stránkování [4, 5]. Virtuální prostor, někdy též nazývaný logický adresový prostor (LAP), je rozdělen na stejně velké stránky (*pages*). Fyzický adresový prostor (FAP) je rozdělen na rámce (*frames*) o stejné velikosti jako stránky. Velikost stránek (= rámců) je některá z mocnin čísla 2. Obvykle to bývá 1 KiB až 16 KiB. Nicméně na architektuře Intel lze zapnout i stránky o velikosti 2 MiB až 4 MiB, což umožňuje adresovat na 32 bitové architektuře větší virtuální adresový prostor.

Nezbytnou součástí stránkování je stránkovací tabulka (*page table*). Ta zajišťuje mapování stránek na rámce. Stánky tvoří souvislý virtuální adresový prostor, ale korespondující rámce mohou být libovolně umístěné ve fyzické paměti. Navíc ne všechny rámce musí být nezbytně mapované do rámců. Následující obrázek 2.1 zobrazuje obecnou představu, jak může vypadat struktura stránkovací tabulky.



Obrázek 2.1: Reprezentace stránkovací tabulky (umístěné ve fyzické paměti) [3]

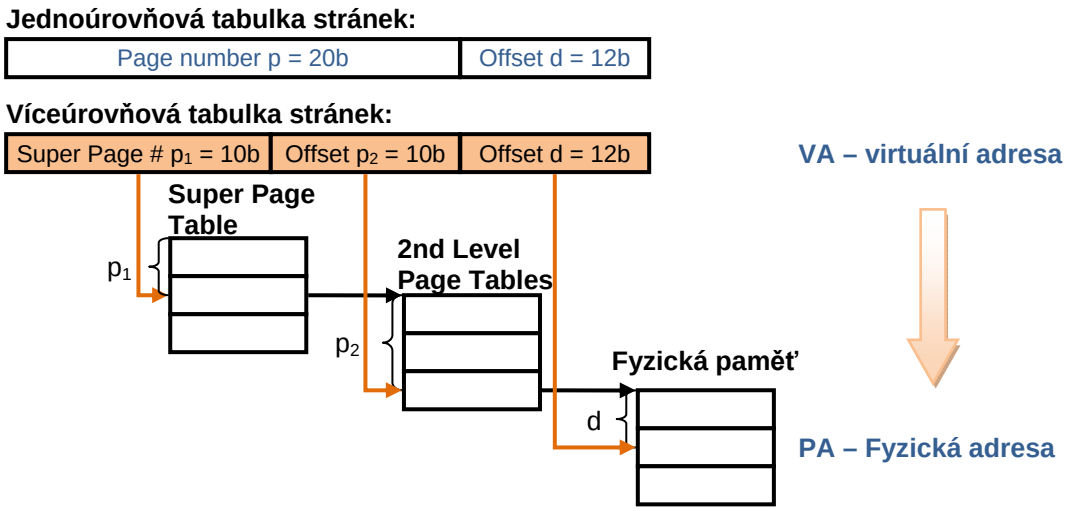
Ve skutečnosti tato struktura bývá složitější a je dána strukturou operačního systému a většinou je předepsána přímo cílovou architekturou, která díky tomu může provádět hardwarovou akceleraci. U moderních multiprogramových operačních systémů je běžné, že tato tabulka není pouze jedna. Obvykle se pro každý běžící proces vytváří vlastní tabulka, díky čemuž může proces přistupovat k celému LAP prostoru. Přesto tomu tak nebývá a část tohoto virtuálního prostoru je sdílena operačním systémem. Děje se tak proto, aby mohl OS jednoduše přistupovat k datům právě běžícího procesu, který volá jádro operačního systému. Pokud by tomu tak nebylo, potom by OS musel přistupovat k datům uživatelského procesu explicitně přes jeho stránkovací tabulku, která by na většině procesorů nemohla být akcelerována, protože není zavedena v registrech procesoru. Proto např. u operačního sytému Windows má standardně každý proces přístup pouze k horním 2 GiB virtuální paměti, což je o polovinu méně než by teoreticky bylo možné.

2.1.1 Víceúrovňové stránkování

Jelikož stránkovací tabulka zajišťuje funkci virtuální paměti, musí být zajištěno, aby byla celá ve fyzické paměti. Jistě si dokážeme představit případ, kdy stránka, která obsahuje část stránkovací tabulky, je odložena na disku. Potom by mohlo dojít k nekonečnému cyklu v nahrazovacím algoritmu, který by se snažil tuto stránku obnovit, ale informace k jejímu obnovení by byly ve stránce, která je také odložena. Navíc by to ani nebylo efektivní, jelikož každá instrukce vyžaduje překlad adres.

Předpokládejme, že každá stránka je reprezentována daty o velikosti 4 B. Tato data mohou být např. ukazatelem na nějakou složitější strukturu, ale to v tomto příkladě není podstatné. Pokud uvažujeme virtuální paměť o velikosti 4 GiB (32 bitový adresový prostor) a stránky o velikosti 4 KiB, dojdeme k zajímavému číslu velikosti výsledné stránkovací tabulky. Jedná se přesně o 4 MiB paměti a když k tomu přičteme fakt, že každý běžící proces musí obsahovat tuto tabulku, tak je tato velikost značný problém. Pokud bychom mluvili o 64 bitovém adresovém prostoru, tak by bylo toto řešení dokonce technicky nemožné.

Ve skutečnosti je stránkovací tabulka velice řídká a používá se pouze malé procento z její celkové velikosti. Jedno z řešení je víceúrovňová tabulka stránek. Tuto techniku používá architektura Intel a její princip je vysvětlen na obrázku 2.2.



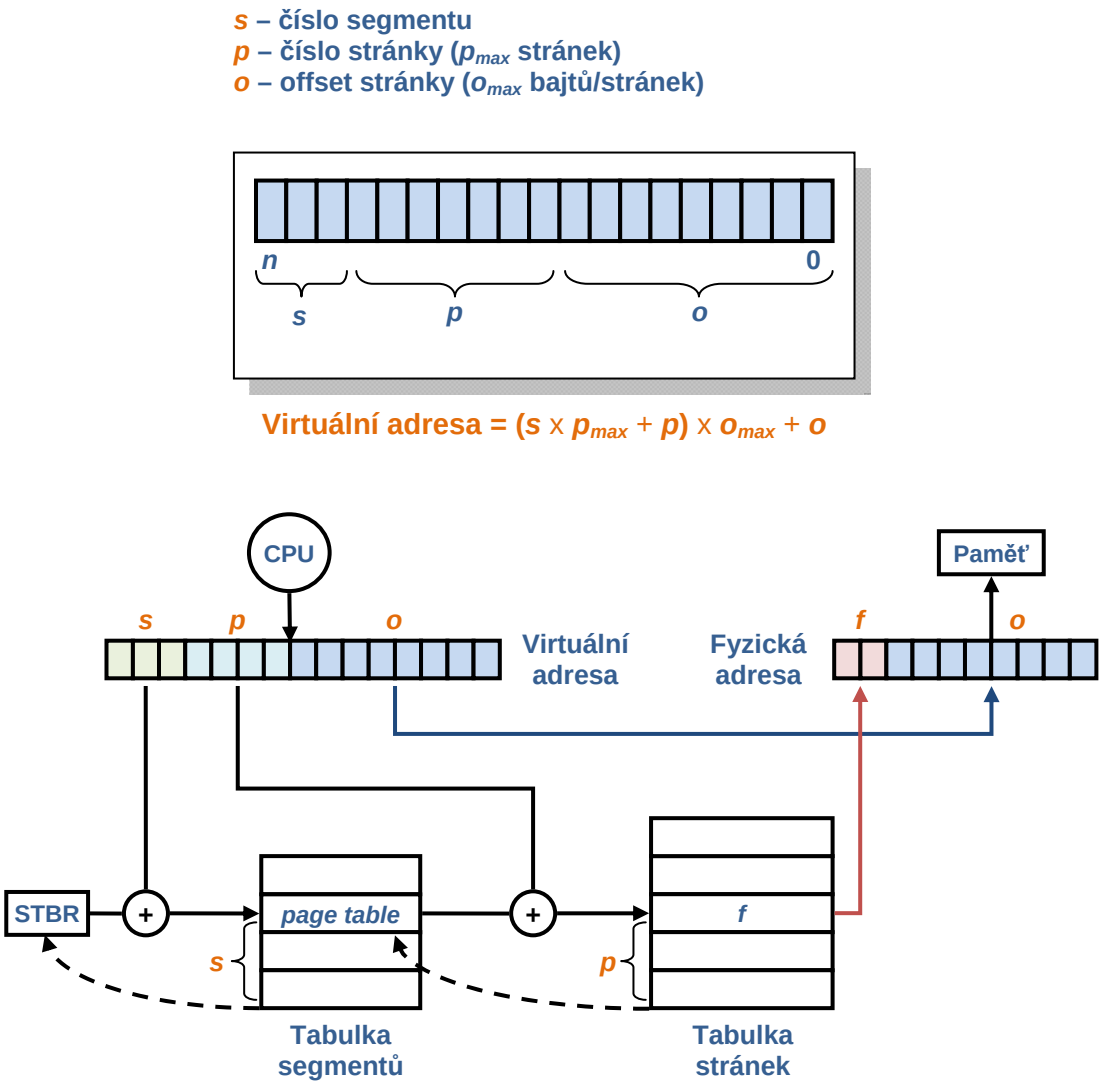
Obrázek 2.2: Výpočet adresy při použití víceúrovňové tabulky [3]

V příkladu je použito dvouúrovňové stránkování, což vyžaduje 3 přístupy do paměti. Tabulka první úrovně obsahuje pouze ukazatele na tabulky druhé úrovně. Velikost tabulky první i druhé úrovně je dána počtem bitů, jejichž počet odpovídá příslušné podčásti logické adresy. V našem případě obě části představují 10 bitů. Pokud předpokládáme, že tabulka je pole ukazatelů o velikosti 4 B, potom velikost odpovídá přesně velikosti stránky, která činí 4 KiB. Tento fakt je velice výhodný i pro systémové správce virtuální paměti. Navíc je toto uspořádání vhodné i díky tomu, že není nutné hledat souvislý blok fyzické paměti a lze jednotlivé tabulky různě rozmístit.

Pokud to nebylo z předchozího textu zřejmé, tak snížení využité paměti je docíleno tím, že tabulky druhé úrovně se vytváří pouze pro platné položky tabulky první úrovně. Pokud bychom ale uvažovali, že bude tabulka obsahovat 100% platných vstupů, tak bychom zjistili, že tato struktura zabírá o 4 KiB více prostoru. Ve skutečnosti by to bylo ještě více, protože každá alokovaná tabulka může v systému představovat další datové struktury. Nicméně, takové extrémní případy jsou velice ojedinělé, a proto je toto uspořádání ve většině případů vyhovující.

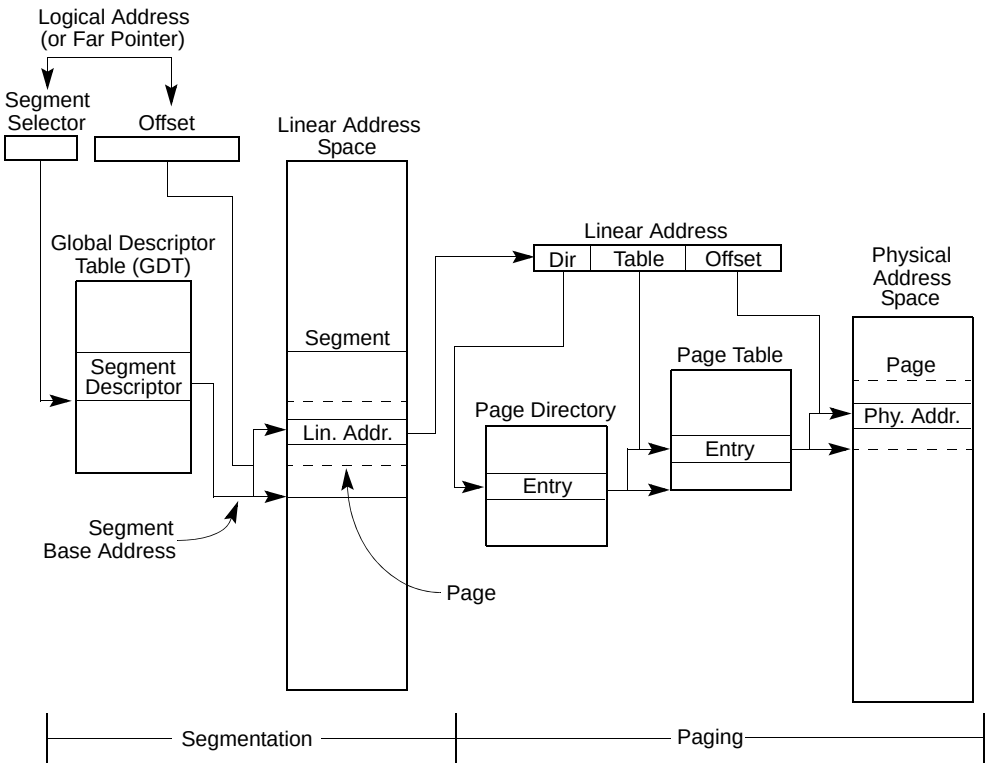
2.1.2 Segmentace

Alternativou ke stránkování [4] je segmentace. Segmentace je adresovací technika operační paměti, která umožňuje zobrazit virtuální paměťový prostor (tzv. virtuální paměť) do fyzického adresového prostoru operační paměti [6]. Mapování adresového prostoru se provádí po segmentech, které mohou mít téměř libovolnou velikost (často zarovnanou na nějaký násobek velikosti dvou, např. 4 byty). Segment obvykle není náhodně vybranou částí paměti, ale má nějaký význam – např. segment obsahující kód programu, kód knihovny, data nebo segment obsahující zásobník procesu.



Obrázek 2.3: Výpočet adresy při použití segmentace [3]

Segmentace nebyla původně vztažená ke stránkování, ale v mnoha systémech je spojená právě s ním. Děje se tak z důvodu lepší specifikace ochrany paměti a jednodušší správy těchto celků na logické úrovni systému.



Obrázek 2.4: Segmentace a stránkování [7]

Pro lepší představu fungování segmentace a stránkování zde uvedu konkrétní příklad implementace v procesorech Intel. Technické prostředky správy paměti architektury IA-32 jsou rozděleny do dvou částí: segmentace a stránkování [7]. Segmentace poskytuje mechanismus oddělení kódu, dat a zásobníku a díky tomu může na jednom procesoru běžet několik programů (úloh) bez vzájemného ovlivňování. Stránkování naproti tomu poskytuje mechanismus pro implementaci tradičního stránkování na žádost (*demand paging*) a systému virtuální paměti. Stránkování může být také použito k vzájemné izolaci některých úloh. U procesorů Intel pracujících v chráněném režimu je možné stránkování zakázat a používat pouze některou z forem segmentace. Použití segmentace se nicméně nedá žádným způsobem vypnout.

Na obrázku 2.4 je zobrazeno, jak segmentace umožňuje rozdělit celý adresovatelný paměťový prostor (nazývaný *lineární adresový prostor*) na malé chráněné adresové úseky zvané *segmenty*. Každému běžícímu programu lze přidělit vlastní množinu segmentů, a tak zajistit vzájemnou izolaci dat. Segmenty mezi programy lze i sdílet, což nám dává možnost sdílení kódu knihoven.

Všechny segmenty leží v lineárním adresovém prostoru procesoru. K tomu, abychom nalezli ve specifickém segmentu konkrétní byte, musíme poskytnout *logickou adresu*. Logická adresa se skládá ze selektoru segmentu a offsetu. Selektor segmentu je unikátní identifikátor segmentu a mimo jiného poskytuje offset do tabulky deskriptorů (např. do globální tabulky deskriptorů GDT), kde jsou uloženy deskriptory segmentů. Každý segment má deskriptor segmentu, který specifikuje vlastnosti segmentu: velikost, přístupová práva, úroveň privilegií, typ a pozici prvního bajtu v lineárním

adresovém prostoru (nazývanou bázová adresa segmentu). Pozice bajtu uvnitř segmentu se následně určí jako součet offsetu s bázovou adresou segmentu. Bázová adresa plus offset tak tvoří lineární adresu v lineárním adresovém prostoru procesoru.

Pokud není použito stránkování, lineární adresový prostor procesoru je přímo mapován na fyzický adresový prostor procesoru. Fyzický adresový prostor je definován jako rozsah adres, které procesor dokáže generovat na své adresové sběrnici.

Při stránkování je každý segment rozdělen na stránky (typicky má každá velikost 4 KiB), které jsou uloženy buď ve fyzické paměti, nebo na disku. Operační systém udržuje strukturu stránek v podobě víceúrovňové tabulky, podobně jak jsme si představili v minulé kapitole. První úroveň tabulky se nazývá adresář stránek (*page directory*) ve které jsou odkazy na množinu tabulek stránek (*page tables*). Pokud program (nebo úloha) chce přistupovat na určitou adresu v lineárním adresovém prostoru, procesor použije adresář stránek a tabulky stránek k tomu, aby přeložil tuto lineární adresu na fyzickou adresu a provedl požadovanou operaci (čtení nebo zápis) na tomto místě.

Požadovaná stránka, která se aktuálně nenachází ve fyzické paměti, vyvolá pomocí výjimky (*page-fault exception*) přerušení běhu programu. Operační systém pak musí zajistit, aby příslušnou stránku načetl do fyzické paměti a transparentně provedl znovuspuštění úlohy v bodě přerušení. Tento princip si obecně vysvětlíme v následující kapitole.

2.1.3 Výpadek stránky

Hlavní funkce stránkování se projeví, když se program snaží přistupovat ke stránce, která se aktuálně nenachází ve fyzické paměti [8]. Tato situace způsobí výpadek stránky (*page-fault*):

1. Zpracování výpadku stránky způsobem neviditelným pro program, který výpadek způsobil a přebrání kontroly řízení.
2. Určení umístění dat v pomocné paměti.
3. Určení rámce ve fyzické paměti vhodného k uložení dat.
4. Pokud byla stránka, aktuálně uložena ve vybraném rámci, od načtení modifikována (je nastaven „dirty“ bit), ulož stránku zpět do pomocné paměti.
5. Načtení požadovaných dat do připraveného rámce.
6. Návrat kontroly řízení programu tak, že se transparentně znovu vykoná instrukce, která způsobila výpadek paměti.

Potřeba odkazovat se na specifickou adresu paměti vzniká ze dvou hlavních důvodů:

- Procesor se snaží nahrát a spustit samotnou instrukci programu.
- Data jsou zpřístupňována instrukcí programu.

Pokud se v kroku 3, kdy musí být nalezen vhodný rámec k uložení dat, nepodaří nalézt volný rámec z důvodu plného využití fyzické paměti, musí být některá existující stránka nahrazena (*swapped*) novou požadovanou stránkou. Ideální stránkovací systém musí určit nahrazovanou stránku tak, aby

nebyla v budoucnu dlouho používána. Existují různé implementace, které se snaží přinejmenším aproximovat tento princip.

Mnoho operačních systémů používá nahrazovací algoritmus LRU (**L**east **R**ecently **U**sed). Teorie za tímto algoritmem spočívá v tom, že stránky, které byly v poslední době nejméně používány, nebudou používány ani v blízké době. Pokud je tedy vyžadována nová stránka, je vyřazena nejméně používaná stránka. Tento algoritmus přináší většinou správné výsledky, ale ne vždy. Např. může existovat proces, který se sekvenčně pohybuje po paměti, ale nikdy opětovně nepoužije naposledy použitou stránku. Následuje stručný přehled nejběžnějších nahrazovacích algoritmů [9] (*page replacement algorithms*):

- **OPT** (Optimal) – nahraď tu stránku, která nebude v budoucnu dlouho používána. Tento teoretický algoritmus byl původně vytvořen pod jménem MIN (Belady, 1969) nyní známý spíše jako OPT (Mattson, 1970). Tyto algoritmy jsou však stěží implementovatelné, protože vyžadují pohled do budoucna. Jejich použití je proto pouze v teoretické rovině ke srovnávání ostatních algoritmů.
- **FIFO** (First in, first out) – nahraď nejstarší stránku. Ta ovšem může být stále používána, tato strategie proto není příliš efektivní. Je ale jednoduchá na implementaci.
- **LRU** (Least Recently Used) – nahraď nejdéle nevyužívanou stránku. Vychází z *principu lokality* – (prostorové) je vysoká pravděpodobnost, že následující odkaz bude směřovat do stejné nebo sousední stránky; (časové) jistý časový interval jsou používány opakovaně stejné proměnné/stejný kód. Při přesné implementaci vyžaduje náročné udržování informací o používaných stránkách. Strategie LRU patří do tzv. zásobníkových algoritmů, tj. algoritmů, u nichž nikdy nemůže vést zvýšení počtu bloků paměti k zvýšení výpadků stránky.
- **LFU** (**L**east **F**requently **U**sed) – nahraď nejméně využívanou stránku. Při přesné implementaci stejně jako LRU vyžaduje náročné udržování informací o používaných stránkách. Může způsobit odstranění právě zavedené stránky, což není ideální.
- **Pseudo-LRU** – označuje skupinu algoritmů podobných LRU s omezenou (často drasticky) přesností. V praxi se často používá právě Pseudo-LRU, případně se stopami LFU. Jedním z představitelů je algoritmus **NRU** (**N**ot **R**ecently **U**sed), který se snaží aproximovat LRU pomocí jednoho bitu. Tato implementace je výhodná díky podpoře na straně HW.

Většina moderních operačních systémů také umožňuje *zamykání stránek*. Pokud je některá stránka označena jako zamčená, nemůže být nahrazena a tedy ani nikdy odložena do pomocné paměti (na disk). Tato vlastnost se s výhodou používá k ochránění citlivých dat, které by se jinak mohli uložit bez vědomí uživatele na disk. Další použití je u kritických systémových dat a realtimových aplikací.

2.1.4 Zavádění stránek na žádost

Zavádění stránek na žádost (*demand paging*) je klasický předpoklad ke studiu a návrhu algoritmů vyrovnávacích pamětí [10], které se snaží pouze při výpadku přesunout stránku z pomalé paměti do rychlé vyrovnávací paměti (*cache*). Zavádění stránek na žádost předchází spekulativní dopředné zavádění stránek (*pre-fetching*). Demand paging se zajímá pouze o jedinou otázku: pokud je cache plná a musí se vložit nová stránka, která stránka musí být nahrazena? Nahrazovací algoritmy, které jsme si popsali v minulé kapitole, se používají obdobně i na úrovni vyrovnávacích pamětí.

Aby dopředné zavádění stránek dokázalo odhadnout budoucí přístup k datům, je nutné znát předchozí historii přístupů. Přesnost odhadu hraje důležitou roli ve snížení „znečištění cache“ a zvýšení účinnosti dopředného zavádění. Přesnost obecně závisí na množství zaznamenané historie přístupu k datům. Řada prací se zaměřila právě na zkoumání tohoto problému, např. [11] použity modely diagramů vztahů, [12] použito schéma odhadu založené na klasickém algoritmu teoretické informatiky Lempel-Ziv, [13] použito schéma založené na asociativní paměti, a [14] použito schéma založené na rozdělení modelovaného kontextu. Komerční nástroje přesto zřídkakdy používají velice sofistikované schémata predikce. Je to způsobeno tím, že k tomu, aby sofistikované metody pracovaly efektivně, je potřeba uchovávat rozsáhlou historii přístupů ke stránkám. Díky tomu jsou tyto metody pro reálné systémy velice těžkopádné a drahé na údržbu.

2.1.5 Ochrana paměti

Paměť, která náleží nějakému procesu, je implicitně u většiny moderních operačních systémů chráněna pomocí virtuálního adresového prostoru [15]. Takové systémy mimoto většinou poskytují i způsob jak explicitně nastavit různé typy ochrany uživatelem nebo systémem alokovaných bloků virtuální paměti. Například lze nastavit blok paměti s kódem na ochranu typu Read-only, a tak ochránit modifikaci vlastního kódu. Těchto vlastností lze využít i při implementaci detekce přetečení paměti. Tento problém podrobněji popíšeme v samostatné kapitole.

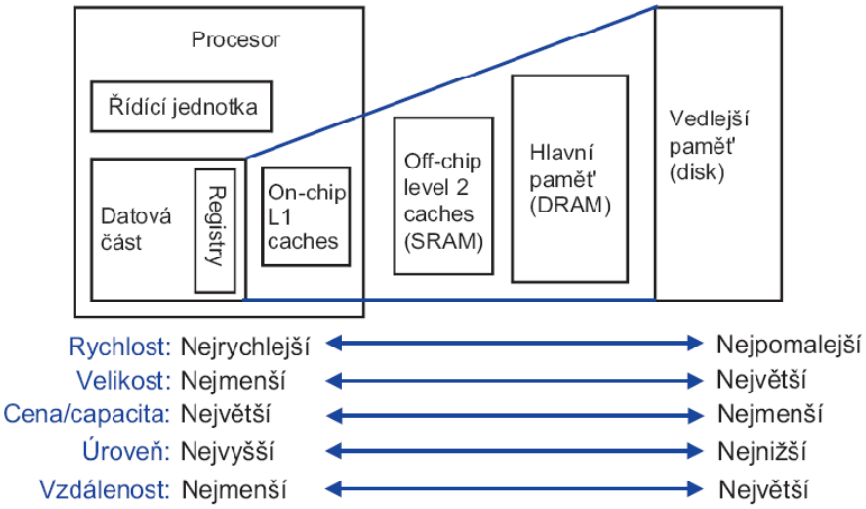
Techniku *Copy-on-write* (COW) [16] zde uvádím proto, že většina systémů tuto vlastnost implementuje jako reakci na porušení ochrany paměti a následné zpracování. Ve skutečnosti se jedná spíše o optimalizaci, která umožňuje více procesům sdílet určitý úsek fyzické paměti, dokud jeden z procesů nezmodifikuje nějakou stránku. Jakmile se některý proces pokusí o zápis, vyvolá se výjimka, kterou zpracuje OS a pokud je nastaven příznak COW, vytvoří pro daný proces kopii této oblasti. Této technice se říká líné vyhodnocování (*lazy evaluation*).

2.2 Prostředky správy paměti poskytované OS

Jelikož výčet funkcí poskytovaných operačními systémy Windows a Linux má výrazně popisný charakter, umístil jsem tento text do příloh na závěr této práce.

3 Paměťová hierarchie

Hierarchické uspořádání prostředků obecně sloužících k uložení dat se u současných počítačových architektur nazývá paměťová hierarchie [17]. Je navržena tak, aby s výhodou využila princip lokality počítačových programů, o kterém si vzápětí napíšeme. Každá úroveň v hierarchii se vyznačuje tím, že má vyšší rychlost, menší velikost a nižší zpoždění než nižší úroveň. Následující obrázek 3.1 tuto vlastnost názorně zobrazuje:



Obrázek 3.1: Paměťová hierarchie reálného počítače [3]

Když procesor přistupuje k paměti, požadavek nejdříve směřuje do primární vyrovnávací paměti (L1 cache). Pokud jsou požadovaná data nalezena v této paměti, nastal zásah (*cache hit*). Pokud v této paměti data nalezena nejsou, nastává *cache miss* a dotaz se musí směřovat o úroveň níž, tedy do L2 cache. Takto se dotaz může obecně propagovat přes všechny úrovně, dokud nejsou data nalezena. Jelikož jsou spolu propojeny vždy jen bezprostředně následující úrovně, musí se nalezená data zpětně propagovat přes všechny úrovně až do nejvyšší primární cache.

I přesto, že je primární paměť relativně malá, dokáže obsloužit mnoho paměťových požadavků. Důvodem je specifické chování programu, které se nazývá princip lokality. Tento princip se dá zjednodušeně vysvětlit tak, že v nějaké krátké době přistupují programy k relativně malé části adresového prostoru. Existují dva typy principů lokality:

1. **Časová lokalita**: pokud byla data použita, je vysoce pravděpodobné, že stejná data budou použita znovu (smyčky, výsledky operací atd.). Proto se snažíme umísťovat nejpoužívanější položky paměti co nejblíže k procesoru.
2. **Prostorová lokalita**: pokud byla použita nějaká data, dá se očekávat, že brzy budou použita data na blízké adrese (instrukce, pole, atd.). Proto se umísťují větší bloky paměťových položek jdoucích za sebou do vyšší úrovně paměti.

3.1 Vyrovnávací paměť (cache)

Nyní se dostáváme k důvodu, proč jsem tuto kapitolu zařadil do mé práce. I když to na první pohled nemusí být úplně zřejmé, můžeme i správcem paměti zefektivnit funkci vyrovnávacích pamětí, které jsou implementovány na úrovni hardwaru. Je ale nutné představit základní funkci těchto pamětí. Dané téma je samo o sobě poměrně rozsáhlé a nekladu si za cíl popsat všechny aspekty problému. Chci především nastínit pozadí problematiky k pochopení barvení cache, která slouží především ke zlepšení činnosti L2 cache. Existuje i metoda barvení celých stránek, kterou např. používá v jádře operační systém FreeBSD [18], ale touto problematikou se nebudeme zabývat. Operační systém Linux používá uvnitř jádra pouze barvení cache a nehodlá to měnit, protože by to přineslo větší složitost správy virtuální paměti.

Jak jsme si již v úvodu naznačili, cache představuje rychlou paměť s malou kapacitou zařazenou mezi procesor a hlavní paměť. Realizuje se jako statická paměť s dobou přístupu pohybujících se v řádech nanosekund (1-10 ns). Pro její realizaci je nutná asociativní paměť. Jako klíč v asociativní paměti se používá adresa položky v hlavní paměti.

Pro jednoduchost si představme vyrovnávací paměť o velikosti 8 KiB. Aby byla podpořena prostorová lokalita, je posloupnost bytů seskupena do jednoho řádku vyrovnávací paměti, tzv. *cache line*. Velikost takového řádku bývá nějaká mocnina dvou nízkého řádu, nejčastěji to je 16, 32 nebo 64 B. Informace přenášené mezi vyrovnávací pamětí a hlavní pamětí je ve smyslu naplnění řádku vyrovnávací paměti, spíše než konkrétního bytu. Pokud tedy program potřebuje určitý byte, je získán z hlavní paměti celý řádek, obsahující tento byte. Předpokládejme, že naše hypotetická cache se skládá z řádků o velikosti 16 bajtů. Zvolená velikost cache 8 KiB nám tak dává přesně 512 řádků a bloky paměti začínající na adrese dělitelné číslem 16 mohou být umístěny do těchto řádků. K tomu, aby mohl systém správně zpracovat požadavky na paměť, potřebuje znát, které bloky paměti jsou nahrané v cache. Navíc tyto informace musí být přítomny v hardwaru, protože je cache zcela kontrolována hardwarem. O tom, jak jsou takové informace mapovány v cache, si napíšeme v následujících podkapitolách. Příklad provázející tento text byl publikován v článku [19].

3.1.1 Plně asociativní

Tento způsob mapování je na první pohled velice výhodný, ale jak v zápětí zjistíme, tak to zcela ideální řešení není. Pokračujme v příkladu a předpokládejme, že pracujeme v 32 bitovém fyzickém adresovém prostoru. Jelikož je hlavní paměť rozdělena do bloků o velikosti 16 bajtů, stačí nám k jednoznačné identifikaci nejvýznamnějších 28 bitů adresy. Zbývající 4 bity slouží pro určení polohy bytu v tomto bloku. Jedním ze způsobu jak určit nahraný blok v řádku cache je plně asociativní vyrovnávací paměť, která vyžaduje na každý řádek 28 bitový *tag registr*, který obsahuje adresu bloku uloženého na tomto řádku. Pokud je vyžadován přístup k hlavní paměti, hardware může porovnat nejvýznamnějších 28 bitů fyzické adresy vůči všem 512 tag registrům. Toto porovnání probíhá

souběžně pomocí 512 komparátorů. Pokud existuje shoda u jednoho z těchto komparátorů, je signalizován *cache hit*. V zápětí se použijí poslední 4 bity adresy k výběru vlastního bajtu.

Bohužel vyrobit plně asociativní paměť je značně složité, protože je potřeba mnoho komparátorů a počet bitů v tag registru je velký. Na tomto příkladu to není možná tak zřejmé, ale u současných L2 cache se již jejich velikost pohybuje v megabytech a muselo by se použít až desítky tisíc komparátorů. O plně asociativní paměti můžeme hovořit např. u TLB (**T**ranslation **L**ookaside **B**uffer), což je také jedna ze specializovaných cache, která se nachází přímo na procesoru a slouží k uložení informací o naposledy použitých stránkách. Používá se jako vyrovnávací paměť tak, aby nemusel pro každou adresaci probíhat výpočet umístění stránky.

3.1.2 Přímo mapovaná

Jedním ze způsobů jak zmenšit problém plně asociativní vyrovnávací paměti, je omezit množinu bloků paměti, které mohou být uloženy na každém řádku cache. Toho může být dosaženo tak, že se hlavní paměť rozdělí na bloky o velikosti 8 KiB. Prvních 16 bajtů ze všech bloků, začínajících na adrese dělitelné číslem 8192, může být nahráno do prvního řádku cache. Dalších 16 bajtů může být nahráno do druhého řádku atd. V takovém případě je potřeba uchovávat pouze, který 8 KiB blok paměti náleží jednotlivým řádkům vyrovnávací paměti. Pozice 16 bajtového bloku uvnitř 8 KiB bloku odpovídá číslu řádku vyrovnávací paměti. Tomuto způsobu sledování korespondence mezi blokem paměti a číslem řádku cache se říká přímo mapovaná vyrovnávací paměť. Zde je fyzická adresa rozdělena na tři části – tag, index a byte offset (viz obrázek 3.2):

Obrázek 3.2: Fyzická adresa pro přímo mapovanou cache [19]

Byte offset se používá pro výběr bajtu z řádku cache. Index je použít pro výběr řádku cache. Tak může obsah každého 16 bajtového bloku obsadit pouze jeden řádek cache. Když je vyžadován přístup k paměti, hardware vybere příslušný řádek vyrovnávací paměti pomocí indexu obsaženého ve fyzické adrese a porovná 16 bitový tag přidělený k tomuto řádku. Pokud dojde ke shodě, nastal cache hit. Pomocí takto získaného indexu je přečten obsah příslušného řádku cache a pomocí 4 posledních bitů fyzické adresy se přečte požadovaný byte.

Přímo mapovaná cache je oproti předchozí metodě nepochybně jednodušší na realizaci, protože je potřeba pouze jeden komparátor a počet bitů tagu je také menší. Protože určité 16 bajtové bloky hlavní paměti mohou obsadit pouze jediný řádek cache, existují případy, kdy opakované výpadky mohou způsobit zpomalení vykonávaného programu. Tyto případy nastávají především u náhodného přístupu do paměti. Pokud se používá sekvenční přístup k paměti, je tato metoda i přes její jednoduchost velice efektivní. Proto je nutné dodržovat určité zásady při tvorbě aplikací, aby

k takovým výpadkům často nedocházelo. Například předpokládejme program, který sčítá dvě pole a do třetího ukládá výsledek. Pokud by pole měly velikost přesně 512x16 bajtů, potom by paměťové adresy obou operandů a také adresa výsledku odpovídala stejnému řádku cache. Tato situace by vedla k výpadkům cache pří každém přístupu do paměti.

3.1.3 S omezeným stupněm asociativity

Existuje kompromisní řešení, které je dnes používáno na všech úrovních cache a tím je cache s omezeným stupněm asociativity (*set-associative cache*). Blíží se k výkonu plně asociativní cache, ale je mnohem jednodušší na realizaci. Kompromis spočívá v posunutí hranice mezi tagem a indexem řádku cache, zobrazeného na obrázku 3.2, doprava. Tím se redukuje počet bitů použitých pro index řádku vyrovnávací paměti, ale počet těchto řádků zůstává nezměněn.

Předpokládejme, že posuneme index řádku o jeden bit doprava a získáme tak pouze 8 bitů, které mají identifikovat číslo řádku cache. Jelikož počet řádků cache zůstává nezměněn, je pochopitelné, že tak již nedokážeme jednoznačně určit všech 512 řádků. Nicméně, takto stále dokážeme určit dvojici řádků, jeden náleží množině obsahujících 256 řádků a další druhé množině. Díky tomu může být každá adresa hlavní paměti umístěna na jednom ze dvou možných řádků vyrovnávací paměti. Pokud tedy uvažujeme pro cache index 8 bitů, pokus o přístup do paměti bude probíhat následovně. Hardware vybere dva řádky na základě indexu získaného z fyzické adresy a souběžně porovná 20 bitový tag fyzické adresy s tag registry příslušející těmto dvěma řádkům. Pokud je na jednom z nich shoda, nastal cache hit a jeden z řádků je vybrán. Na závěr se standardně vybere odpovídající byte pomocí posledních 4 bitů.

Vyrovnávací paměť s takovým chováním se nazývá dvoucestná cache s omezeným stupněm asociativity (*two-way set-associative cache*), protože jsou pomocí indexu řádku vybrány dva řádky cache. Pokud posuneme cache index o další bit, získáme tak čtyřcestnou cache (*four-way cache*), protože index řádku (7 bitový) označuje čtyři řádky cache. Pokud posuneme cache index zcela doprava a tím tento index úplně eliminujeme, získáme tak plně asociativní cache.

Čtyřcestná vyrovnávací paměť se v mnoha aplikacích velice blíží k výkonu plně asociativní vyrovnávací paměti o stejné velikosti. Což umožní efektivnější práci s dynamickými strukturami, jako jsou lineární seznamy, stromy atd., které mají charakter náhodného přístupu do paměti, při pokusech o lineární procházení celé struktury. Příklad popisující sečtení dvou polí a uložení výsledku do třetího, popsaný výše, na této architektuře bude pracovat zcela bez problému. Je to dáno tím, že všechny tři operandy plus instrukce ležící na stejném řádku, se mohou vždy vyhnout výpadku cache. V tomto případě souběžné porovnání tagů vyžaduje pouze čtyři komparátory, každý porovnávající 21 bitů. Protože takové uspořádání přináší mnoho výhody a navíc je stále relativně jednoduše proveditelné, je tento způsob realizace cache v současnosti velice rozšířen. Dokonce se dnes již používá 8cestné mapování pro L2 cache a 16cestné pro L1 cache [20].

3.1.4 Barvení cache

Pro lepší využití cache se na úrovni správy paměti používá technika zvaná barvení cache. Ta vychází z vlastností mapování fyzických adres do vyrovnávací paměti. Jak jsme si již uvedli, výhoda např. dvoucestné cache s omezenou asociativitou spočívá ve faktu, že lze přistupovat na dvě adresy paměti na 8 KiB hranicích a vždy získat dva rozdílné řádky vyrovnávací paměti. Nicméně, pokud je potřeba přistupovat na třetí adresu paměti, která je také na hranici dělitelné 8 KiB, způsobí to konflikt [21].

Problém nastává, když se používají dynamické struktury, které mají za sebou jdoucí prvky různě rozmístěné v paměti. Uveďme si tentokrát jako příklad cache o velikosti řádku 32 bytů a velikost struktury 64 bytů [18]. Alokované struktury jsou zřetězeny v seznamu a tento seznam je procházen. Protože ukazatel na následující prvek máme v prvních 4 bytech této struktury, při projití se bude přistupovat pouze na prvních 32 bytů dané struktury. Pokud by struktury byly alokovány bez barvení těsně za sebou, tak by operace projití seznamu stresovala sudé adresy v L1 cache a vůbec by nevyužívala liché adresy.

Uvedený problém do jisté míry řeší vícecestná cache, ale pokud bychom procházeli přes více hranic paměti, mapovaných do cache, docházelo by k častým výpadkům. Problém se dá řešit relativně snadno tím, že stránku pro alokaci objektů nerozdělíme bezezbytku, ale část paměti rezervujeme pro barvení cache. Čím větší rezervu (*overhead*) vytvoříme, tím získáme více „barev“. Počet barev se získá jako podíl rezervy a velikosti řádku cache. Aby byly všechny adresy cache použity rovnoměrně, objekty v dalších alokovaných stránkách jsou vždy posouvány o velikost řádky cache. Toto posouvání se provádí periodicky v cyklech vycházejícího z počtu barev.

4 Synchronizace

Synchronizace odkazuje na jeden ze dvou navzájem blízkých konceptů a to synchronizaci procesů a synchronizaci dat [22]. Synchronizace procesů spočívá v myšlence, že se několik paralelně běžících procesů na určitém místě spojí, aby mohly dosáhnout nějaké shody nebo provést nějakou sekvenci akcí. Synchronizace dat má zajistit koherenci mezi několika kopiemi dat nebo udržet jejich integritu. Synchronizace procesů se často používá k implementaci synchronizace dat.

Ve vícevláknových aplikacích je obvykle potřeba přistupovat k některým zdrojům z jiných vláken. Takový zdroj potom nazýváme sdílený a musíme zajistit jeho konzistenci. V závislosti na typu zdroje/objektu k němu lze povolit nebo zakázat souběžný přístup. Například není problém, pokud několik vláken čte nějaký globální parametr souběžně. Nicméně, problém nastává, pokud se snažíme současně měnit nějakou složitější sdílenou strukturu, např. lineární seznam. Pokus o přidání prvku do lineárního seznamu totiž vyvolá několik kroků, které by mohly být přerušeny uprostřed jiným vláknem, a při návratu by tato struktura byla v nekonzistentním stavu. U těchto typů operací říkáme, že nejsou atomické (nedělitelné). Atomičnost těchto operací proto musíme zajistit jiným způsobem. K tomu nám operační systémy poskytují řadu synchronizačních nástrojů.

4.1 Synchronizační nástroje

Mezi nejdůležitější pojmy, které se týkají synchronizačních nástrojů, patří kritická sekce. Je to úsek kódu, který přistupuje ke sdíleným prostředkům (datům, strukturám, zařízením, …) a může být v jeden okamžik vykonáván pouze jedním ze souběžně běžících vláken. Předpokladem je, že kritická sekce skončí v konečném čase, jinak by docházelo k blokování čekajících vláken, které do této sekce musí vstoupit také v konečném čase. Kritická sekce nám zajistí atomičnost skupiny operací. K tomu abychom ji realizovali, operační systém poskytuje prostředky (zámek/lock), které se vždy umisťují před (*entry*) a za (*exit*) kritickou sekcí.

Nejobecnější zámek je semafor (Dijkstra 1963). Umožňuje vstup do kritické sekce i více než pouze jednomu vláknu a připouští odemknutí zámku i jinému procesu, než který ho uzamknul. Tato technika se proto používá především pro signalizaci událostí, kde je zajištěno, že se žádná událost nemůže ztratit. Z obecného semaforu lze odvodit binární semafor, který je již vhodný ke vzájemnému vyloučení.

V operačních systémech se používá především *mutex*. Je to speciální binární semafor určený pouze pro vzájemné vyloučení, který může odemknout pouze proces, který ho zamknul. Jeho jméno je odvozeno z anglického výrazu *MUTual EXclusion* – česky vzájemné vyloučrní. Mutex se používá především v meziprocesové synchronizaci, jelikož může být sdílen mezi procesy. Z toho plyne jeho nevýhoda. Při jeho použití je vždy vyžadován přístup do plánovače procesů, který je spuštěn

v kontextu jádra, což vede k přepínání kontextu při každém volání a tím výraznému zpomalování. Tento synchronizační nástroj je proto vhodný pouze k dlouhodobému vyloučení s pozastavením procesu.

Na systému Linux je mutex zobecněn i pro práci s POSIX vlákny a bývá napsán jako knihovní funkce, což umožňuje běh na aplikační úrovni, a tak je mnohem vhodnějším pro použití rychlé synchronizace vláken. Na rozdíl od Linuxu používá systém Windows speciální objekt nazývaný kritická sekce (*critical section object*), který může být použit pouze k synchronizaci vláken běžících v kontextu jediného procesu. Jeho výhodou je především větší efektivita, která je zajištěna použitím specifických instrukcí procesoru *Test and Set*. A proto nedochází k volání jádra v případech, kdy byl zámek volný.

4.1.1 Spinlock

Speciální kategorií synchronizačního nástroje je *spinlock*, který také slouží ke vzájemnému vyloučení, ale na rozdíl od předchozích metod používá aktivní čekání. To ho předurčuje ke krátkodobému vyloučení vláken běžících na víceprocesorových systémech. Aktivní čekání značí, že je vlákno stále aktivní a procesor v cyklu čeká na uvolnění zámku z jiného vlákna, které běží na jiném procesoru. Pokud je zámek zamčen pouze na krátkou dobu, stane se tato synchronizace velice účinnou, jelikož nedochází k pozastavení vláken a přepínání kontextu, vyžadující nemalou režii. Přepnutí vlákna může také vést k zneplatnění dat v L1 cache procesoru. Tímto se celá operace změny kontextu vlákna ještě více prodraží. Na jednoprocesorových systémech takový druh synchronizace postrádá smysl, protože se kontext musí stejně vždy ze zřejmých důvodů přepnout (neexistuje jiný procesor, který by dokázal zámek odemknout) a pouze by docházelo k mrhání procesorového času.

Spinlock je třeba obezřetně používat i na procesorech, které podporují HT (**H**yper **T**hreading) [23]. Tato technologie se používala u některých řad procesorů Intel a mnohdy byla chápána jako vícejádrový procesor. Nicméně, technologie HT používá stále jedno jádro mající x samostatných stavových jednotek. Pokud bychom na takovém procesoru použili spinlock, tak zaměstnáme jádro procesoru natolik, že vlákno v druhé logické jednotce procesoru vůbec nedostane šanci něco provést. Výsledek je pak stejný jako na jednom procesoru bez HT.

Na systému Windows existuje rozšíření synchronizačního objektu kritická sekce, které ho umožní používat jako spinlock. Kritická sekce se musí před jejím použitím inicializovat pomocí funkce *InitializeCriticalSectionAndSpinCount()*. Tato API funkce sice přináší několik vylepšení v porovnání s funkcí *InitializeCriticalSection()*, přesto by se ale měla používat obezřetně. Problém nastává, jak jsme si již v úvodu popsali, u jednoprocesorových systémů.

Systém Linux prostřednictvím POSIX vláken také umožňuje použít tento druh zámku. Lze použít funkce *pthread_spin_lock()* a *pthread_spin_unlock()*. Ale opět musíme být ze stejných důvodů obezřetní.

Jiný problém dotýkající se výkonu aplikace při použití zámků je *lock convoy*. Tento problém nastává, pokud několik vláken se stejnou prioritou opakovaně žádá o stejný zámek. Na rozdíl od uváznutí (*deadlock*) nebo blokování, vlákna stále běží. Nicméně, pokaždé když se vlákno pokouší získat zámek a selže, tak se vzdá zbytku svého naplánovaného časového kvanta a vynutí si přepnutí vlákna. Režie opakovaného přepínání kontextu a nedostatečné využití naplánovaného časového kvanta sníží celkový výkon. Tento problém je typický u běžně užívaných sdílených prostředků chráněných kritickými sekcemi, mezi něž patří právě i správce paměti.

Kritické sekce ve Win32 jsou nejen levnější než ostatní synchronizační objekty, ale jsou také více odolné vůči convoy problému [23]. Operační systém totiž zná vlastníka kritické sekce. To mu umožňuje dočasně zvýšit prioritu vlastníka a tím mu dát více prostoru k uvolnění zámku. Windows 2003 přináší ještě jedno drobné vylepšení pro kritické sekce. V předchozích verzích Windows kritická sekce propouštěla vlákna v pořadí FIFO. Ve Windows 2003 jsou čekající vlákna ve frontě vybírána v náhodném pořadí. Pokud všechna vlákna čekající v *EnterCriticalSection()* provádějí stejné operace, tak tato změna nepřinese žádné zásadní zlepšení. Pokud ale provádí každé vlákno něco jiného, tak se tímto způsobem zlepší odolnost kritické sekce vůči convoy problému.

Ačkoli můžeme použít standardní prostředky operačních systémů k synchronizaci, ukazuje se, že lze napsat uživatelský zámek, který je mnohem obecnější a dokonce i efektivnější. Prostudováním daného problému jsem vytvořil následující zámek:

```
void Lock(SPINLOCK * pSpinLock)
{
    if (TryLock(pSpinLock)) return;
    if (GetNumberOfCPUs() < 2) {
        while (!TryLock(pSpinLock)) SwitchToThread();
        return;
    }
    int dwSpinCount = (int)(SPIN_COUNT * GetSpinFactor());
    int dwSpinning  = dwSpinCount;

    while (--dwSpinning >= 0 && *pSpinLock == LOCKED) YieldProcessor();

    if (*pSpinLock == LOCKED) {
        SwitchToThread();
    }

    while (!TryLock(pSpinLock)) {
        dwSpinCount = (int)(dwSpinCount * 0.75);
        dwSpinning  = max(min(MAX_SPINS,dwSpinCount), MIN_SPINS);

        while (--dwSpinning >= 0) {
            if (TryLock(pSpinLock)) return;
            YieldProcessor();
        }
        Sleep(1);
    }
}
```

```
static double GetSpinFactor()
{
    static double factors[] = { 0.809, 1.050, 0.951, 0.758, ... };
    return factors[GetThreadId()%(sizeof(factors)/sizeof(double))];
}

#define TryLock(pLock) (                                                \
    *(pLock) == UNLOCKED &&                                             \
    InterlockedCompareExchange(pLock,LOCKED,UNLOCKED) == UNLOCKED       \
)

void Unlock(PSPINLOCK pSpinLock)
{
    MemoryBarrier();
    *pSpinLock = UNLOCKED;
}
```

Předchozí kód je psaný pro systém Windows, ale po malých úpravách lze bez větších zásahů použít i na systému Linux. Nicméně, tento systém neobsahuje funkci *InterlockedCompareExchange*, která je zásadní pro uvedený druh zámku. Používá totiž nativní instrukci procesoru pro realizace atomické operace *Compare-and-swap*. Ta odpovídá následujícímu pseudokódu:

```
int CompareAndSwap(int *v, int old, int new) atomic
{
    if (*v == old) { *v = new; return 1; }
    else return 0;
}
```

Jelikož nám nic nebrání použít vkládaný assembler, můžeme funkci pro konkrétní procesor dopsat sami. Např. procesory Intel používají od řady 486 instrukci `CMPXCHGB`.

Samotný kód zámku je napsaný tak, aby mohl být beze změny použit na jednoprocesorovém, víceprocesorovém případně s procesorem používající HT. U všech druhů procesorů se nejdříve zkusí rychlý test, zda-li je zámek volný pomocí makra TryLock. To nejdříve rychle otestuje, jestli je otevřený a nesnaží se okamžitě použít atomickou instrukci procesoru, která může na úrovni hardwaru vyvolat synchronizaci procesorů. Pokud je zámek volný, tak se ho pokusí získat a okamžitě přejít do kritické sekce. Když je zámek vlastněn jiným vláknem, tak se dál postupuje podle druhu procesoru. U jednoprocesorových nemá smysl použít spinning, jak jsem již psal. Proto je v cyklu vynucováno okamžité přepnutí vlákna do té doby, než se vláknu podaří získat zámek. Pokud se jedná o víceprocesorový systém, je použit spinning. Funkce *GetSpinFactor()* slouží jako heuristika. Snahou je získat různé hodnoty čítače pro různá vlákna. Tato vlastnost ale nemá až takový vliv na celkový výkon, jelikož jistou náhodu délky cyklu přidává samotný operační systém i použitý hardware. Po získání délky nastává vlastní spinning. Pro procesory s HT slouží pseudofunkce *YieldProcessor()* jako nápověda pro procesor, že vykonávaný kód běží v kontextu spinlocku. Moderní procesory Pentium obsahují instrukci `PAUSE`, která umožňuje optimalizovat chování procesoru v případech, kdy je použit spinning pro získání zámků držených různými vlákny. Pokud zámek zůstal stále zamčený, je nejdříve provedeno rychlé přepnutí vlákna. Poté je znovu proveden pokus o získání

zámku, a pokud opět selhal, začne se při každém neúspěchu o čtvrtinu snižovat spinning, až se dostane na své minimum. Testy ukázaly, že je nejlepší spinning po chvíli zcela potlačit. Nejdříve jsem v cyklu používal jako u jednoprocesorového systému funkci *SwitchToThread()*. Nicméně to přináší problém v podobě *lock convoy*. Musíme si uvědomit, že přepnutí vlákna vyvolá plánovač, který se snaží spustit všechna naplánovaná vlákna na všech dostupných prostředcích. Pokud by na daném procesoru běželo tolik vláken, kolik je dostupných jader, tak by přepnutí vlákna nezpůsobilo jeho pozastavení, ale vedlo by to k jeho okamžitému naplánování a opětovnému spuštění na stejném nebo jiném jádru. Díky tomu by byl procesor vytížen na jeho maximum a vedlo by to k naprosté degradaci výkonu. Proto je mnohem výhodnější použít funkci *Sleep()*, pomocí které skutečně pozastavíme vlákno. Toto pozastavení ale nesmí být na příliš dlouhou dobu, aby nebyla vlákna zbytečně dlouho v nečinnosti. Jedna milisekunda už stačí k tomu, aby byl procesor vytížen na přijatelnou mez a celkový výkon byl na vysoké úrovni. Jednotlivé výsledky, kterých jsem při testech dosáhnul, popíši v samostatné kapitole věnované výhradně srovnávacím testům.

4.1.1.1 Rekurzivní spinlock

Pokud jsme navrhli vlastní synchronizační nástroj, je vhodné, aby byl schopný také fungovat jako rekurzivní zámek. Tento termín značí problém, kdy se opakovaně snažíme získat zámek ve stejném kontextu provádění. Jelikož zámek může odemknout pouze vlastník, došlo by k uváznutí. Na systému Windows jsou všechny synchronizační nástroje standardně rekurzivní, u systému Linux se musí speciálně nastavit nebo nejsou podporovány vůbec. Pokus o opětovné přidělení zámku ve stejném kontextu obvykle značí špatný návrh aplikace, ale v některých situacích se bez něj nelze obejít. Následující rozšíření umožní používat obecně libovolný zámek rekurzivně:

```
typedef struct {
      LOCK        lock;             // klasický zámek
      int         dwThreadID;       // jednoznačný identifikátor vlákna
      int         dwCount;          // počet rekurzivního volání zámku
} RECURSIVE_LOCK, *PRECURSIVE_LOCK;

void RecursiveLock(PRECURSIVE_LOCK pRLock)
{
      if (pRLock->dwThreadID == GetThreadID()) {
            pRLock->dwCount++;
      } else {
            Lock(&pRLock->lock);
            pRLock->dwThreadID = GetThreadID();
      }
}

void RecursiveUnlock(PRECURSIVE_LOCK pRLock)
{
      if (pRLock->dwCount == 0) {
            pRLock->dwThreadID = INVALID_THREAD_ID;
            Unlock(&pRLock->lock);
      } else {
            pRLock->dwCount--;
      }
}
```

Z přiloženého kódu si lze povšimnout, že se nepoužívá žádná dodatečná synchronizace, která by zajistila atomičnost těchto operací. Nicméně je nutné, aby získání hodnoty `dwThreadID`, při pokusu o zamčení, bylo atomické. Pokud bude proměnná ležet na zarovnané adrese procesoru, tak předpoklad bude splněn, jelikož základní instrukce to zajišťují.

Jelikož není vhodné navrhovat synchronizaci, aniž bychom měli algoritmus ověřený, použil jsem verifikační nástroj Spin založený na jazyku Promela. Spin je nástroj pro analýzu logické konzistence paralelních systémů. Systém je popsán v modelovacím jazyku Promela (*Process Meta Language*). Spin během simulace a verifikace kontroluje bezpečnost, uváznutí, živost apod.

K simulaci zámků jsem vytvořil následující kód v jazyce Promela:

```
int lock = 0;
int dwThreadID = NPROC;
int dwCount = 0;

#define spin_lock(mutex)                                \
      do                                                \
      :: 1 -> atomic {                                  \
            if                                          \
            :: (mutex == 0) -> mutex = 1; break;        \
            :: else -> skip;                            \
            fi;                                         \
      }                                                 \
      od

#define spin_unlock(mutex)                              \
      mutex = 0


#define rspin_lock()                                    \
      if                                                \
      :: (dwThreadID == _pid) ->                        \
            dwCount++;                                  \
      :: else ->                                        \
            spin_lock(lock);                            \
            hThread = _pid;                             \
      fi


#define rspin_unlock()                                  \
      if                                                \
      :: (dwCount == 0) ->                              \
            dwThreadID = NPROC;                         \
            spin_unlock(lock);                          \
      :: else ->                                        \
            dwCount--;                                  \
      fi
```

Se zámky jsem posléze provedl několik simulací, které neodhalily žádný problém, a tudíž můžeme implementaci považovat za bezpečnou. Do tohoto textu nelze umístit veškeré zdrojové kódy, a proto jsou dostupné na přiloženém CD.

5 Manuální správa paměti

Při manuální správě paměti má programátor plnou kontrolu nad tím, zda a ve kterém okamžiku bude paměť uvolněna a případně využita pro opakované přidělení [17]. To obvykle nastává buď explicitním voláním funkcí pro přidělování a uvolňování paměti z hromady (např. malloc/free v jazyce C), nebo jazykovými konstrukcemi ovlivňujícími zásobník (např. pro lokální proměnné). Klíčovou vlastností manuální správy paměti je možnost, aby program sám vrátil část paměti a oznámil, že ji již dále nepotřebuje. Bez tohoto oznámení správce žádný úsek opakovaně nevyužije.

Výhody manuální správy paměti [18]:

- Pro programátora může být snazší pochopit přesně, jak funguje.
- Některé manuální alokátory mají lepší výkon při nedostatku paměti.

Nevýhody manuální správy paměti:

- Programátor musí opakovaně psát hodně kódu k údržbě záznamů o alokované paměti.
- Správa paměti tvoří významnou část rozhraní libovolného modulu/knihovny.
- Manuální správa typicky vyžaduje vyšší paměťovou režii na každý objekt.
- Chyby ve správě paměti jsou velmi časté.

Potom, co jsem napsal nevýhody manuální správy paměti, se může zdát, že je tento způsob zastaralý a v dnešní době nepoužitelný. Je pravda, že automatická správa paměti je čím dál více používána, ale přesto se u kritických aplikací, jako jsou operační systémy, servery, realtimové aplikace, atd., stále používá manuální správa. Je to dáno tím, že se na dané úrovni používají především strukturované jazyky a programátoři vyžadují naprostou kontrolu nad přidělováním a uvolňováním paměti. Mimoto algoritmy použité pro manuální správu mohou sloužit jako základ automatické správy.

Pro programátory je velmi běžné, z důvodu neefektivní nebo nevhodné manuální alokace, že si píší vlastní kód, který kopíruje chování správy paměti. Např. tak, že si alokují větší blok paměti a k použití ji rozdělí. Takový kód je známý jako subalokátor [18]. Subalokátory mohou využít výhody znalosti chování programu, ale jsou obecně méně efektivní než změna základního alokátoru.

5.1 Fragmentace paměti

Fragmentace je jev, který vede v mnoha formách k plýtvání paměťového prostoru [19]. Existují tři různé ale související termíny: interní fragmentace, externí fragmentace a fragmentace dat. U správy paměti jsou nejdůležitější první dva termíny a jsou spolu velmi těsně svázány [20]. Často zjistíme, že pokud zmenšíme interní fragmentaci, tak zvýšíme externí fragmentaci a naopak. Řada správců paměti trpí na jeden nebo více z těchto nedostatků a v důsledku toho snižují celkovou kapacitu paměti.

5.1.1 Interní fragmentace

Interní fragmentace vzniká, když správce paměti alokuje pro každou alokaci více, než je vyžadováno. Tím vzniká nevyužitá část přiděleného úseku [18]. Existují tři důvody, proč tento jev vzniká:

1. **Výplň** (*padding*) – obvykle se vkládá kvůli zarovnání adresy paměti na pole objektů nebo na objekt samotný.
2. **Buddy system** – jedná se o podtřídu alokátorů s oddělenými seznamy. Výhodou tohoto alokátoru je velká rychlost při alokování a uvolňování bloků paměti. Nevýhodou je právě velká interní fragmentace.
3. **In-band hlavičky** – někteří správci paměti alokují větší blok paměti, než je potřeba a jistou pevnou velikost z tohoto bloku použijí např. k uložení informací o velikosti bloku nebo k uložení tagů. Tato dodatečná paměť se také někdy nazývá rámec.

5.1.2 Externí fragmentace

Špatně navržená metoda přidělování paměti může vést k tomu, že nelze přidělit dostatečně velký blok volné paměti, i když celkové množství volné paměti je větší. Tato situace vzniká tehdy, pokud je volná paměť rozdělena na mnoho malých bloků, mezi nimiž jsou stále používané bloky.

5.1.3 Fragmentace při stránkování a segmentaci

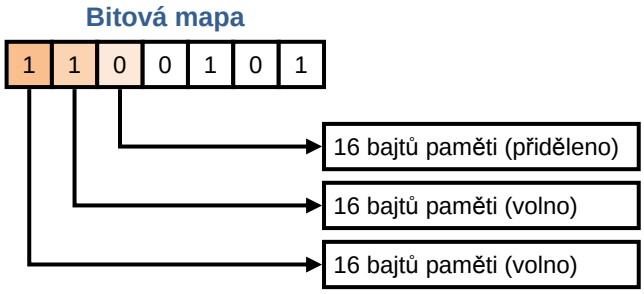
V první kapitole jsme si popsali stránkování a segmentaci paměti na nejvyšší úrovni správy paměti. Pro zajímavost v této kapitole uvedu srovnání obou metod z pohledu fragmentace.

Při použití stránkování se eliminuje externí fragmentace [20], neboť každý rámec může být přidělen k nějaké stránce a v operační paměti nezůstává část, kterou by nebylo možné přidělit. Nadále však existuje interní fragmentace, neboť rámce jsou přidělovány jako celé jednotky a pokud požadavek na přidělenou paměť nebude právě roven některému z násobků velikosti rámce, bude poslední přidělený rámec zaplněn pouze zčásti. Z hlediska interní fragmentace jsou vhodnější pokud možno menší stránky. Na druhé straně je však třeba si uvědomit, že se zmenšováním počtu stránek nutně roste velikost datové struktury tabulky stránek.

Při použití segmentace se odstraní interní fragmentace, neboť délka segmentu je právě rovná požadovanému paměťovému prostoru procesu a navíc se může měnit. Na rozdíl od stránkování se však opětovně objevuje externí fragmentace. Přidělení paměťového bloku jednomu segmentu probíhá obdobně, jako je tomu u přidělování dynamicky tvořených souvislých úseků paměti (např. algoritmy typu *best-fit*, *first-fit*). Poté, co je paměťový blok uvolněn, vzniká prázdná oblast (díra). Pro snížení externí fragmentace tedy lze použít techniku setřásání v některé z jejích podob.

5.2 Alokátory s bitovou mapou

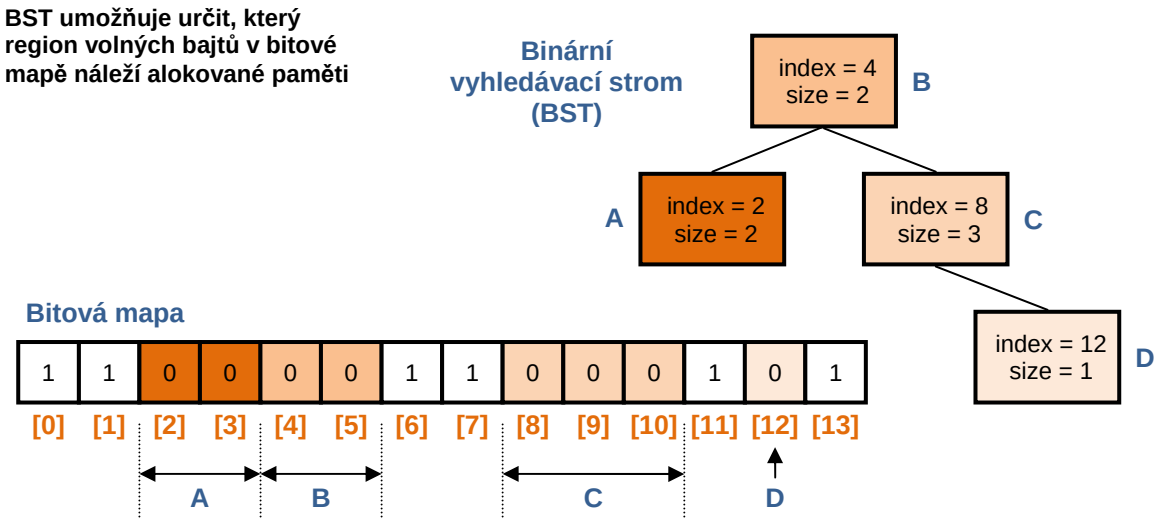
Tento přístup používá pole bitů k udržení záznamů o tom, které oblasti paměti jsou volné a které obsazené. Paměť je rozdělena na malé úseky o stejné velikosti a každý bit v bitovém poli reprezentuje jeden z úseků [21]. Jak je ilustrováno na obrázku 5.1.



Obrázek 5.1: Bitová mapa spravovaných bloků paměti

Problém tohoto přístupu je, že bitová mapa neobsahuje, kolik paměti bylo alokováno pro danou oblast. Jediné, co nám bitová mapa může poskytnout, je, které regiony paměti jsou volné a které obsazené. Proto nejsme schopni alokovanou paměť správně uvolnit.

To znamená, že potřebujeme rozšířit bitovou mapu o další strukturu, která umožní uchovat, kolik paměti je rezervováno pro každou alokaci. Jedno z řešení je použít nějaký druh vyhledávacího stromu. Tyto dvě datové struktury se vzájemně doplňují. Bitová mapa je použita během alokace k nalezení volného paměťového regionu a vyhledávací strom během uvolňování k určení kolik bitů bitové mapy se musí vynulovat (viz obrázek 5.2).



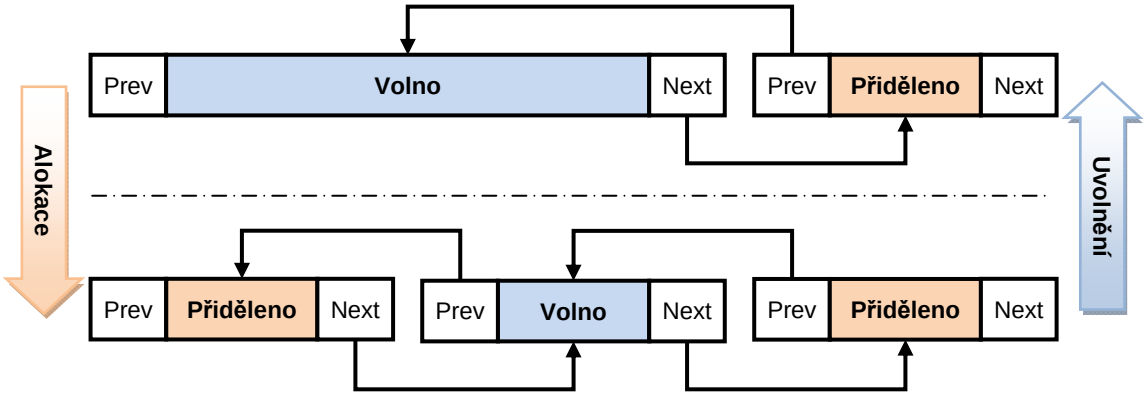
Obrázek 5.2: Nastavený bit (1) označuje volný region a smazaný bit (0) obsazený region.

Správa paměti s bitovou mapou je poměrně nenáročná na implementaci a může minimalizovat interní fragmentaci paměti. Mnoho dřívějších operačních systémů používalo bitmapy jako podpůrný prostředek při správě paměti. Bylo to z důvodu pevné velikosti bitmap, což umožňovalo umístění alokátoru mimo chaotickou dynamickou paměť jádra. Další z výhod je relativně malá velikost bitmapy ve srovnání s tím, jak velkou oblast paměti dokáže popsat. Např. pokud každý bit reprezentuje 16 bajtů paměti, bude potřeba 512 KiB bitmapy ke správě 64 MiB paměti. Ačkoli velikost neobsahuje prostor potřebný k uložení vyhledávacího stromu, je vidět, že metoda je velice úsporná na paměť.

Na druhou stranu může být problém v nalezení souvislé oblasti volných bitů v bitmapě. V nejhorším případě musíme projít celé bitové pole, což zapříčiňuje velmi drahou cenu alokace z pohledu času. S použitím paralelních instrukcí, které obsahují moderní procesory, a bitových masek lze tento čas poměrně snížit. Přesto tato metoda nepatří k nejrychlejším.

5.3 Rodina alokátorů „Sequential Fit“

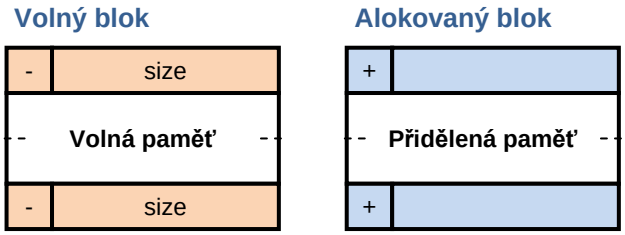
Technika sekvenčního vyhledávání vhodného bloku (*sequential fit*) [21] organizuje paměť do lineárních seznamů volných a rezervovaných regionů (viz obrázek 5.3). Pokud je vyžadována alokace, správce paměti sekvenčně projde seznamem, dokud nenalezne volný blok paměti, který vyhovuje (*fit*) požadované velikosti. Podle toho, jakým způsobem se prochází seznam volných bloků, jsou nazvány jednotlivé druhy algoritmů – *first fit*, *next fit*, *best fit*, atd.



Obrázek 5.3: Správa volných a obsazených bloků paměti

Typicky není zapotřebí externí datové struktury, protože část alokované paměti lze rezervovat pro účely indexace sama sebe. Logické uspořádání bloků paměti lze vidět na obrázku 5.3. Tento způsob je podobný Knuthově metodě „boundary tag“ [22]. Ta byla navržena tak, aby spojení dvou sousedních volných bloků proběhlo se složitostí O(1). V uvedené metodě jsou rezervována dvě slova na blok, která vytváří lineární seznam. Základní idea je, že udržujeme velikost volného bloku zakódovanou ve slově na obou hranicích volného bloku. Díky tomu můžeme vždy zjistit velikost

bloku samotného i pozici a velikost bloků okolních. S těmito informacemi je již spojení bloků poměrně snadné. Každý blok musí navíc obsahovat příznak (uvnitř hraničního slova), zda-li je volný nebo obsazený (viz obrázek 5.4).



Obrázek 5.4: Knuthova metoda „boundary tag“

Alokace paměti je poměrně přímočará. Správce paměti určitým způsobem prochází lineární seznam paměťových bloků, dokud nenalezne volný blok, který je dostatečně velký na to, aby uspokojil požadavek. Pokud je volný blok větší, než byl požadavek, musí se rozdělit na dvě části. Jedna část se použije k obsloužení požadavku a zbývající zůstane volná k obsloužení požadavků následujících. Algoritmus pro uvolňování bloků paměti vyžaduje spojení přilehlých bloků. Pro blok umístěný mezi dvěma jinými bloky existují čtyři různé scénáře spojení. Spojování probíhá tak, aby nikdy nemohla nenastat situace, kdy by vedle sebe ležely dva volné bloky. Při rozdělování a spojování bloků se musí vždy správně upravit ukazatele NEXT a PREVIOUS nebo hraniční slova, pokud používáme metodu „boundary tag“. Na obrázku 5.3 je vidět první z těchto způsobů.

5.3.1 First fit

Metoda výběru prvního vhodného bloku [17] jednoduše prochází seznam volných bloků a vybere z něj první blok, jehož velikost je větší nebo rovna požadované velikosti. Je-li blok větší, rozdělí se a zbývající část je vložena zpět do seznamu. To však vede k situaci, že dělením velkých bloků na začátku seznamu vznikne mnoho malých bloků, jejichž sekvenční procházení může podstatně zpomalit operaci přidělování paměti. Jedním z možných řešení je využití složitějších datových struktur pro ukládání volných bloků, např. stromů.

5.3.2 Next fit

Modifikací metody first fit je výběr dalšího volného bloku (*next fit*) [17], kdy vyhledávání vhodného bloku začíná vždy na pozici, kde předchozí vyhledávání skončilo. Prohledávání tedy nezačíná vždy na stejném místě, ale postupně prochází celým seznamem, což zabrání hromadění menších bloků na začátku seznamu. Velkou nevýhodou této metody je však to, že bloky přidělované v téže fázi výpočtu mohou být od sebe značně vzdáleny, z čehož pak vyplývá menší lokalita odkazů a s ní spojený pomalejší přístup do paměti. Naopak bloky s různou dobou života mohou ležet vedle sebe, a to zase způsobuje větší fragmentaci paměti poté, co jsou bloky s kratší dobou života uvolněny. Celkově lze tedy říci, že i přes některé výhody vede next fit k menší efektivitě přidělování paměti.

5.3.3 Best fit

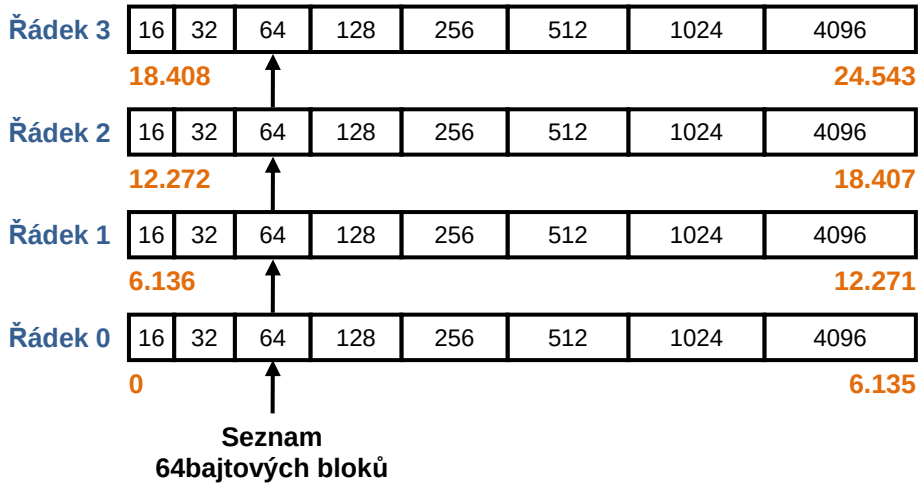
Principem této metody je vyhledání nejmenšího volného bloku paměti, jehož velikost je větší nebo rovna požadované [17]. To vede k minimalizaci ztrát zajištěním menší fragmentace paměti, ovšem pokud bude shoda délek příliš velká, ale ne dokonalá, vznikne velké množství většinou nepoužitelných malých zbytků paměti. Při prohledávání volných bloků musíme navíc projít všechny (pokud nenajdeme přesnou shodu), takže pro velké množství objektů bude prohledávání, a tedy i celé přidělování paměti, pomalé. Proto se používají spíš varianty založené například na vyvážených binárních stromech.

Jedním z možných vylepšení tohoto algoritmu je využití segregované paměti s více seznamy volných bloků, kdy jsou v jednom seznamu umístěny bloky přibližně stejné délky. Při vyhledávání nejlepšího vhodného bloku se pak prochází kratší seznam. Další možností je indexované seřazení bloků podle délky, ať již opět do lineárních seznamů, nebo do složitějších datových struktur.

5.4 Alokátory s oddělenými seznamy

U sekvenčního vyhledávání bloků jsme se hodně věnovali rozdělování a spojování bloků paměti. Také jsme zjistili, že lineární procházení celého seznamu nemusí být vždy nejrychlejší. Alokátory s oddělenými seznamy [21] se tyto problémy snaží řešit tak, že udržují několik seznamů každý s pevnou velikostí bloků. Jinými slovy, je hromada (*heap*) rozdělena na několik skupin bloků podle jejich velikosti. Pokud potřebujeme např. blok o velikosti 32 bajtů, zjistíme, zda-li příslušný seznam obsahuje nějaký prvek (volný úsek paměti) místo toho, abychom procházeli celý seznam.

Jedna z nejjednodušších implementací je rozdělení paměťového prostoru na předem zvolené třídy velikostí bloků. Na obrázku 5.5 je rozdělena paměť na řádky o velikosti 6.135 bajtů. Každý řádek sestává z osmi různých paměťových bloků. Velikost prvního elementu v každém řádku je 16 bajtů a velikost posledního 4.096 bajtů.



Obrázek 5.5: Rozdělení paměti na řádky a vytvoření různých tříd

Jelikož seznam elementů má pevnou velikost a jejich pozice je také známa, není potřeba takovou strukturu udržovat v lineárním seznamu. Nicméně, je potřeba udržovat u každého elementu hlavičku se stavem o minimální velikosti jednoho bajtu. Stav bude udržovat informace o tom, zda je blok volný nebo obsazený. Pokud je obdržen požadavek na alokaci, nejprve se prohledají elementy, které se nejvíce blíží požadované velikosti. Jestli v této třídě již neexistuje žádný volný element, pokračuje se postupně do tříd o větší velikosti, dokud se nenajde nějaký volný blok.

V porovnání s technikami „sequential fit“ je tento algoritmus nesrovnatelně rychlý, ale za cenu většího plýtvání s pamětí. Skutečnost, že tato technika poskytuje pouze určité, předem dané velikosti bloků, vede k interní fragmentaci paměti. Navíc tento problém narůstá, pokud se často musí alokovat paměť z vyšší třídy než z té, která je nejblíže požadavku. Může se například stát, že požadavek na 32 bajtů paměti je uspokojen až blokem o velikosti 1.024 bajtů.

Dalším problémem implementace je fakt, že nejsme schopni alokovat blok paměti větší než je maximální velikost elementu, který se nachází na jednom řádku. S tímto problémem se potýkají i mnohem komplikovanější algoritmy. V následujícím textu se zaměříme na ostatní metody, které se v praxi často používají a vychází z myšlenky alokátorů s oddělenými seznamy.

5.4.1 Přidělování s omezenou velikostí bloku (Buddy system)

Tyto metody přidělování paměti jsou založeny na hierarchickém dělení volného paměťového prostoru na části [17]. V nejjednodušším případě je paměť rozdělena na dvě velké části, ty se dále dělí na menší části atd. Toto dělení paměti definuje omezující podmínky na to, kde jsou bloky paměti alokovány, jaká je jejich možná velikost a jak mohou být volné bloky znovu spojovány do větších celků. Pro každou možnou velikost bloku se udržuje samostatný seznam volných bloků, takže jde v podstatě o variantu přidělování výběrem nejlepšího vhodného bloku, i když s nejrůznějšími variacemi týkajícími se způsobu rozdělování a spojování bloků. Zásadou spojování bloků je, že se mohou spojit pouze ty bloky, které spolu sousedí na stejné úrovni hierarchie (*buddies*). Výsledný blok pak patří do bezprostředně vyšší úrovně hierarchie.

Cílem tohoto systému je, aby bylo možné při uvolnění bloku najít jeho souseda pouze jednoduchým adresovým výpočtem. Sousedem může být buď celý volný blok, nebo blok, jenž je přidělen jako celek nebo je rozdělen na další bloky. Jinou výhodou je nízká paměťová režie, postačuje pouze jediný bit obsahující informaci o tom, zda je blok volný nebo ne. Žádné další ukazatele nebo pomocné informace nejsou potřeba. Při uvolňování bloku však musíme znát jeho velikost.

5.4.1.1 Binární přidělování

Nejznámější variantou je binární přidělování [17], kdy jsou velikosti bloků vždy mocninami dvou. Při dělení se blok vždy rozdělí na dvě stejné části. Všechny bloky leží na adrese zarovnané na mocninu dvou, každý bit relativní adresy bloku vzhledem k začátku volné paměti představuje jednu úroveň hierarchie – má-li hodnotu 0, jde o první polovinu, má-li hodnotu 1, jde o druhou polovinu

nadřazeného bloku. Přístup k blokům je tedy založen na jednoduchých výpočtech zvládnutelných pomocí bitových operací.

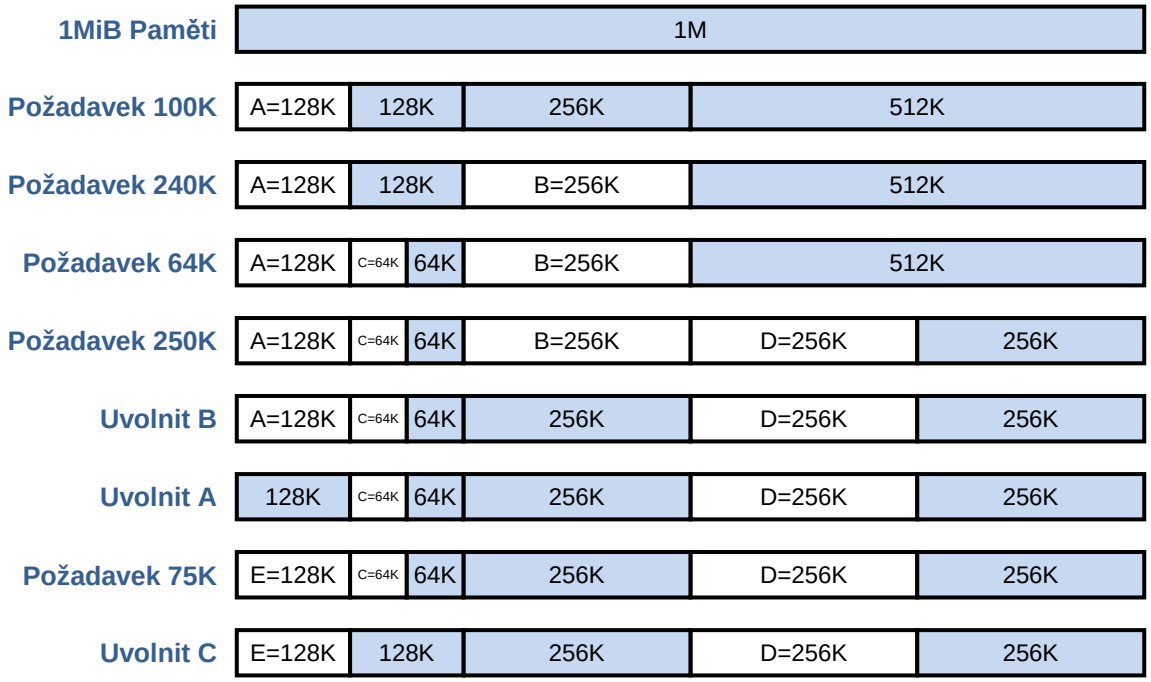
Postup při alokaci paměti [23]:

1. Najdi paměťový slot vhodné velikosti (minimální 2^k blok větší než požadovaná paměť).
   4. Pokud je nalezen, je alokován programu.
   5. Pokud ne, pokus se vytvořit vhodný paměťový slot:
      1. Rozděl volný paměťový slot větší než požadovaný na poloviny.
      2. Pokud je dosažen spodní limit, alokuj tuto velikost paměti.
      3. Vrať se na krok 1 (vyhledání paměťového slotu vhodné velikosti).
      4. Toto opakuj, dokud není nalezen vhodný paměťový slot.

Postup při uvolňování paměti:

1. Uvolni paměťový blok.
2. Podívej se na sousední blok, jestli je také volný.
3. Pokud je, spoj oba a vrať se ke kroku 2. Tento proces opakuj, dokud není dosažen horní limit (veškerá paměť je uvolněna) nebo je nalezen soused, který není volný.

Obrázek 5.6 ukazuje, jak může vypadat alokace a uvolňování paměti při použití buddy systému. V praxi se většinou nejdříve stanoví maximální velikost bloku, kterou alokátor může vrátit a při inicializaci je celkový paměťový prostor na tyto bloky rozdělen. Tento postup je zvolen, protože se pracuje s mnohem většími úseky paměti a je potřeba snížit počet seznamů volných bloků, kterých musí být tolik, kolik je různých tříd velikostí bloků.



Obrázek 5.6: Příklad alokace a uvolňování paměti v buddy systému

5.4.1.2 Fibonacciho přidělování

Varianta Fibonacciho přidělování [17] se snaží snížit vnitřní fragmentaci využitím kompaktnější sady možných velikostí bloků. Vzhledem k tomu, že každý prvek Fibonacciho řady je součtem dvou předcházejících prvků, lze blok vždy beze zbytku rozdělit na dva bloky, jejichž velikosti jsou opět prvky řady. Jistým problémem této metody je, že pokud přidělíme blok určité velikosti, má zbytek po dělení jinou velikost a pravděpodobně nebude příliš užitečný, pokud bude program požadovat přidělení většího počtu bloků téže velikosti.

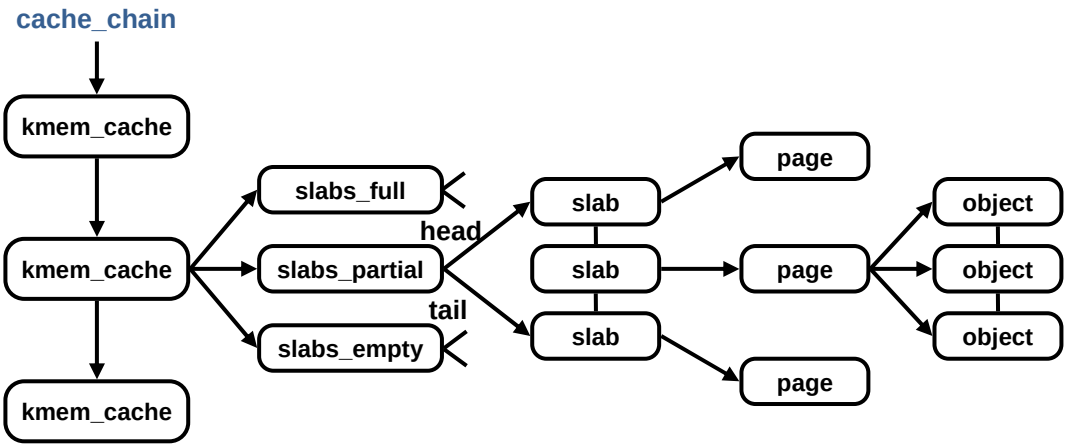
5.4.2 SLAB alokátor

V informatice se algoritmus slab alokátoru využívá k vyřešení problému interní fragmentace paměti a navíc je i velice rychlý, jelikož používá cache objektů [24]. Interní fragmentace vzniká, jak jsme si již popsali, alokováním bloku paměti většího než požadovaná paměť (viz Buddy alokátor). Slab alokátor i buddy alokátor se velmi často společně vyskytují v jádrech různých operačních systémů.

Algoritmus slab alokátoru byl poprvé představen Jeffem Bonwickem v operačním systému SunOS [25]. Tento algoritmus se zaměřil na cachování objektů. V jádru je značné množství paměti alokované pro konečný počet objektů jako například deskriptory souborů a jiné běžné struktury [26]. Bonwick zjistil, že čas potřebný k inicializaci objektu v jádře překračuje čas potřebný k jeho alokaci a dealokaci. Jeho závěr byl, že místo uvolňování paměti do globální paměti (*pool*) je lepší nechat paměť inicializovanou pro konkrétní účel. Následující alokace paměti nepotřebuje vykonat inicializaci, protože je již objekt v požadovaném stavu, který získal při posledním uvolněním voláním dekonstruktoru. Slab alokátor v jádře Linuxu vyšel z těchto a jiných myšlenek, aby vybudoval alokátor efektivní jak prostorově tak i časově.

Slab alokátor také podporuje hardwarové zarovnávání adres a barvení cache, což umožňuje objekty z různých cache umístit do stejných „cache lines“, čímž se zvýší využití hardwarových vyrovnávacích pamětí a tím i výkonu.

Obrázek 5.7 znázorňuje vysokoúrovňový pohled na strukturu slab alokátoru. Na nejvyšší úrovni je cache_chain, což je lineární seznam obsahující několik slab cache. Toto je výhodné pro algoritmus best fit, který se snaží vyhledat cache, která nejlépe vyhovuje požadované velikosti. Každý prvek seznamu cache_chain je odkaz na strukturu kmem_cache (nazývanou cache). Ta ke správě definuje skupinu objektů (*objects pool*) dané velikosti.



Obrázek 5.7: Významné struktury slab alokátoru

Každá cache obsahuje seznam slabů, které jsou tvořeny souvislými bloky paměti (typicky stránkami). Existují tři druhy slabů:

- **slabs_full** – slaby, které jsou plně alokované.
- **slabs_partial** – slaby, které jsou částečně alokované.
- **slabs_empty** – slaby, které jsou prázdné nebo nemají žádný alokovaný objekt.

Slaby nacházející se v slabs_empty jsou prvotní kandidáti na uvolnění (*reaping*). Toto je proces, kdy je paměť využívaná slabem vrácena zpět do systému pro jiné použití.

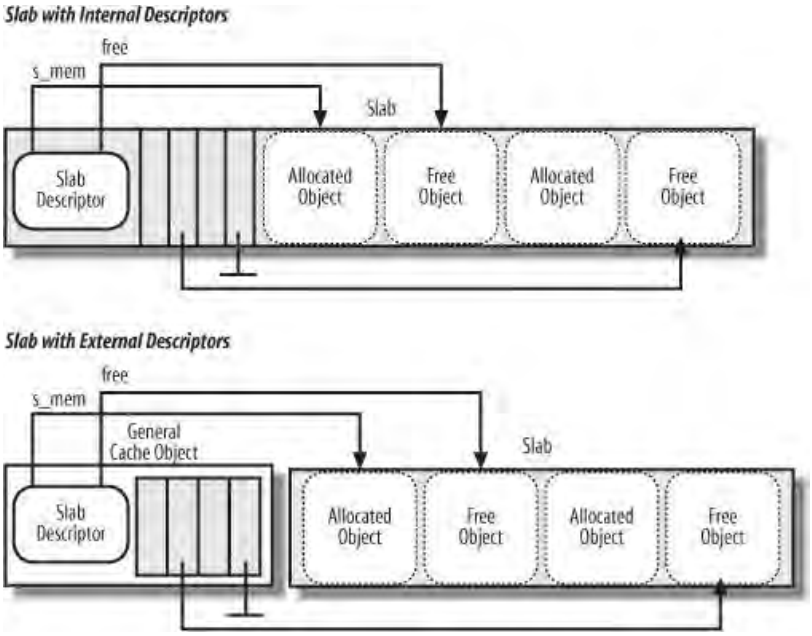
Každý slab v seznamu slabů je souvislý blok paměti (jedna nebo více souvislých stránek), který je rozdělen na objekty. Tyto objekty jsou základní elementy, které jsou alokovány a uvolňovány z jednotlivých cache struktur. Všimněte si, že slab je nejmenší alokovanou jednotkou, kterou slab alokátor používá. Tedy pokud potřebuje růst (*grow*), toto je minimum, o které může vzrůst. Typicky je pro každý slab alokováno četné množství objektů.

Jak jsou objekty alokovány a uvolňovány ze slabu, každý slab se může přesouvat mezi různými seznamy slabů. Například, když jsou všechny objekty z konkrétního slabu spotřebovány, přesune se ze seznamu slabs_partial do slabs_full. Když je slab plný a některý objekt je uvolněn, je přesunut ze seznamu slabs_full do slabs_partial. Pokud jsou všechny objekty uvolněny, přesune se ze seznamu slabs_partial do slabs_empty.

Slab je ve skutečnosti tvořen deskriptorem slabu a deskriptorem objektů [27]. Každý objekt je popsán deskriptorem, což jsou v podstatě jen indexy tvořící lineární seznam volných objektů popř. speciální hodnota, která indikuje přidělení objektu. Deskriptory objektů jsou uloženy v poli umístěného hned vpravo za odpovídajícím deskriptorem slabu. Tyto deskriptory je možné uložit dvěma možnými způsoby, které jsou znázorněny na obrázku 5.8.

- **Externí deskriptor objektů** – uložen mimo slab. Pro uložení těchto deskriptorů se používají obecné cache, které mají různou velikost a jsou vytvořeny v inicializaci slab alokátoru. Jelikož jsou tyto cache pojmenovány podle velikosti objektů, které uchovávají, říká se jim někdy *sizes cache*. Velikost cache, která bude použita pro deskriptor, závisí na počtu uložených objektů vytvářeného slabu.
- **Interní deskriptor objektů** – uložen uvnitř slabu, přímo před objekty, které popisuje.

První deskriptor objektu v poli popisuje první objekt ve slabu atd. Deskriptor objektu je jednoduše bezznaménkové celé číslo, které má význam, pouze když je objekt volný. Obsahuje index dalšího volného objektu ve slabu, a tak tvoří jednoduchý seznam volných objektů uvnitř slabu. Deskriptor objektu posledního elementu seznamu volných objektů je označen nějakou speciální hodnotou. Nejčastěji to bývá nejvyšší hodnota celého čísla (0xffff).



Obrázek 5.8: Vztah mezi slabem a deskriptorem objektů [27]

5.5 Urychlení alokace s využitím cache

V kapitole o hierarchii paměti jsem popsal vyrovnávací paměti na úrovni hardwaru. Na úrovni softwaru se můžeme běžně setkat v jádrech operačních systémů s různými přístupy, které mají charakter vyrovnávacích pamětí a snaží se pomocí predikce přidělovat více prostředků nebo neuvolňovat okamžitě přidělené prostředky s předpokladem, že budou v blízké době potřeba. Tyto techniky se snaží především zefektivnit práci se souborovým systémem nebo s virtuální pamětí. Na aplikační úrovni můžeme naproti tomu snížit frekvenci přístupů do jádra.

Při tvorbě alokátoru potřebujeme ze systému alokovat úsek virtuální paměti, který dále spravujeme a přidělujeme podprogramům na aplikační úrovni. Pokud tento úsek paměti po nějaké době nebude postačovat, musí se alokovat ze systému nový. Bylo by logické, že jakmile potřeby aplikace na přidělenou paměť klesnou, okamžitě přidělený úsek vrátíme systému. Takový přístup může vést v mnoha případech k neustálému volání jádra o přidělení a uvolnění paměti. Proto je vhodné počkat po určitou dobu, kdy tento úsek není používán. Tuto dobu není jednoduché odhadnout, a proto je k parametru nutné přidat ještě informaci o celkovém množství přidělené paměti. Jakmile množství klesne pod určitou hranici, úsek paměti se vrátí systému.

Další důležitou vlastností je omezení synchronizace vláken na minimum. To, že se používají rychlé synchronizační nástroje, ještě neznamená, že nezpomalují celkový výkon aplikace. Díky tomu, že operační systémy umožňují každému vláknu spravovat soukromá data (*thread local data*), ke kterým mají efektivní přístup bez použití synchronizace, lze vytvořit vyrovnávací paměti pro každé vlákno. Princip spočívá v tom, že si každé vlákno z globálního správce paměti, který musí být *thread-safe*, alokuje určitý počet objektů, které uloží do zásobníkové struktury (*cache*) nacházející se v lokálním úložišti vlákna. Pokud je vyžadováno v některém vláknu přidělení paměti, nejdříve se alokátor podívá do cache tohoto vlákna a pokud obsahuje nějaký objekt, je okamžitě z této cache odstraněn a předán vláknu a to zcela bez synchronizace. Jestliže již není v této cache žádný objekt, je opět získán z globálního alokátoru. Naplnění cache probíhá vždy dávkově a je alokováno vždy více objektů naráz. Při uvolňování se zjistí kolik volných objektů je již přichystáno v aktuální cache. V případě, že tento počet nepřevyšuje určitou hranici, je objekt vrácen zpět do této cache, tedy nedojde ke skutečnému uvolnění paměti. Pokud je ale tato hranice překročena, je objekt okamžitě uvolněn pomocí globálního alokátoru. Takto je zajištěno, že si některé vlákno nerezervuje veškerou paměť pouze pro sebe.

5.6 Optimální alokátor

Pokud se vytváří nějaký nový algoritmus, je vždy dobré vědět, jakého výsledku jsme dosáhli. Proto je vhodné vytvořit idealizovaný referenční model, pomocí něhož lze porovnat dosažený výsledek. V tomto odstavci si představíme algoritmus alokátoru, který má vždy složitost O(1) a jeho kód je velice minimalistický. Na jeho základu jsou např. vytvořeny části slab cache. Jeho nevýhodou je, že dokáže alokovat pouze objekty o stejné velikosti. Nicméně tento nedostatek lze snadno odstranit, pokud uvažujeme paměť o dostatečné velikosti.

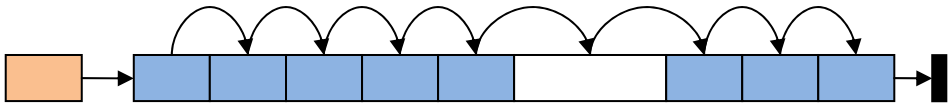
Nejdříve uvedu základní podobu tohoto alokátoru a poté se zmíním o jeho rozšíření. Zaveďme si následující strukturu, kterou potřebujeme pro správnou funkci algoritmu:

```
typedef struct {
    char*       pObjFree;           // ukazatel na první volný objekt
    char        pMemPool[];         // rezervovaná paměť k alokaci
} OPT_ALLOCATOR;
```

Inicializace této struktury má lineární složitost O(n), ale protože je provedena pouze jednou, tak ji při měření nebudeme uvažovat. Mimoto je možné inicializaci přenést do alokace, kdyby docházelo k postupnému rozšiřování paměti a složitost inicializace i alokace by zůstala konstantní. Ale protože chceme zkoumat především maximální rychlost alokace a uvolňování paměti, tak tento alokátor vytvoříme z co nejmenším počtem výpočetních kroků. Vlastní inicializace bude probíhat následovně (obrázek 5.9), z důvodu abstrakce zde nebudeme řešit rezervování paměti ze systému:

```
bool InitOptAllocator(OPT_ALLOCATOR* pOptAlloc, int dwSize, int dwCount)
{
    char* pAddr = pOptAlloc->pObjFree = pOptAlloc->pMemPool;
    for (int  i = 0; i < dwCount-1; ++i,pAddr+=dwSize) {
        *((char**)pAddr) = pAddr+dwSize;
    }
    *((char**)pAddr) = 0;

    return true;
}
```

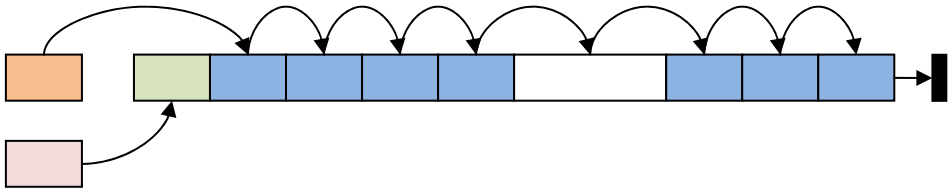


Obrázek 5.9: Stav alokátoru po inicializaci

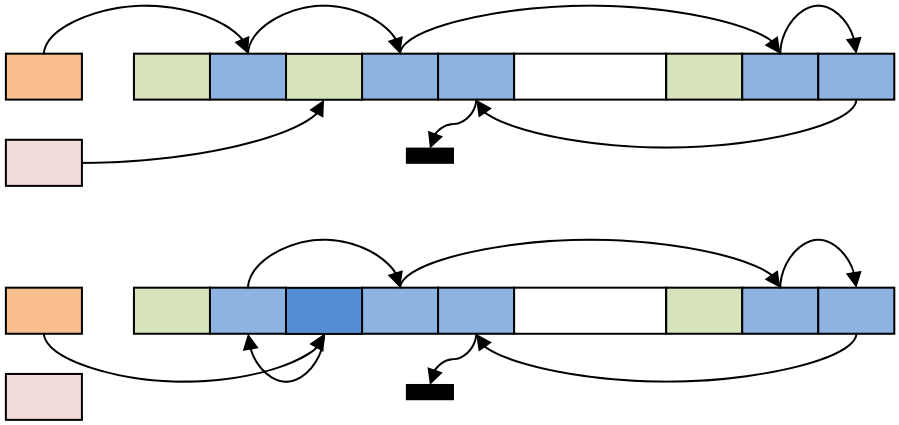
Inicializace vytvoří v podstatě lineární seznam volných objektů, každý o velikosti dwSize. Poslední prvek je nastaven na nulový ukazatel, který zajistí vrácení NULL pokud již není žádný prvek volný. Realizace alokátoru potom vypadá následovně:

```
void* OptAlloc(OPT_ALLOCATOR* pOptAlloc)
{
    char*  pObj = pOptAlloc->pObjFree;
    if (pObj) pOptAlloc->pObjFree = *((char**)pOptAlloc->pObjFree);
    return pObj;
}

void OptFree(OPT_ALLOCATOR* pOptAlloc, void* pObj)
{
    *((char**)pObj) = pOptAlloc->pObjFree;
    pOptAlloc->pObjFree = (char*)pObj;
}
```



Obrázek 5.10: Stav alokátoru po prvním přidělení paměti



Obrázek 5.11: Stav alokátoru po určité době a vrácení paměti

Tento alokátor má nejspíše jednu z nejefektivnějších implementací. A jeho rychlost by se výrazně nezměnila ani, kdyby se postupně rozšiřovala jeho kapacita. Toho bychom mohli dosáhnout následovně:

```
typedef struct {
    int         dwObjSize;          // velikost jednoho objektu
    int         dwMemSize;          // velikost rezervované paměti
    char*       pObjFree;           // ukazatel na první volný objekt
    char        pMemPool[];         // rezervovaná paměť k alokaci
} OPT_ALLOCATOR;


bool InitOptAllocator(OPT_ALLOCATOR* pOptAlloc, int dwSize, int dwCount)
{
    pOptAlloc->dwSize = 0;
    pOptAlloc->pObjFree = 0;
    pOptAlloc->dwMemSize = 0;
    return true;
}

void* OptAlloc(OPT_ALLOCATOR* pOptAlloc)
{
    char* pObj = pOptAlloc->pObjFree;
    if (pObj) {
        pOptAlloc->pObjFree = *((char**)pOptAlloc->pObjFree);
    } else {
        pObj = pOptAlloc->pMemPool + pOptAlloc->dwMemSize;
        pOptAlloc->dwMemSize += pOptAlloc->dwObjSize;
    }
    return pObj;
}
```

Aby bylo možné alokovat různé velikosti, lze vytvořit pole těchto alokátorů. Každé pole bude odpovídat určité velikosti. Indexem do tohoto pole bude požadovaná velikost nebo nejbližší velikost požadované. Otázka vždy zní, jaká interní fragmentace paměti je ještě přípustná.

6 Koncept a implementace efektivního alokátoru

Rozbor jednotlivých alokátorů ukázal, že žádný z nich není naprosto ideální a musí se volit podle toho, jaké vlastnosti v dané aplikaci preferujeme. Navíc ani jeden z nich se v jeho základní podobě nezabývá problémem použití ve víceprocesorových/vícevláknových aplikacích. Mohlo by se zdát, že tyto algoritmy stačí pouze správně rozšířit o synchronizační nástroje a bez problémů je dále používat. Tato myšlenka je sice správná, ale pokud chceme vytvořit efektivní alokátor, zjistíme, že toto jednoduché řešení díky přibývajícímu počtu paralelních úloh řádově snižuje celkový výkon.

K tomu aby byl alokátor skutečně efektivní je potřeba zajistit několik podmínek. Zaprvé je nutné zvolit vhodnou synchronizaci, která se výrazně podílí na celkové době obsloužení požadavku, pokud je současně vyžadován přístup do kritické sekce. Tento problém a jeho řešení jsem popsal v kapitole 4. Zadruhé je nutné vytvořit alokaci dostatečně rychlou, aby vlákna, snažící se vstoupit do kritické sekce, dlouho aktivně nečekala ve spin locku.

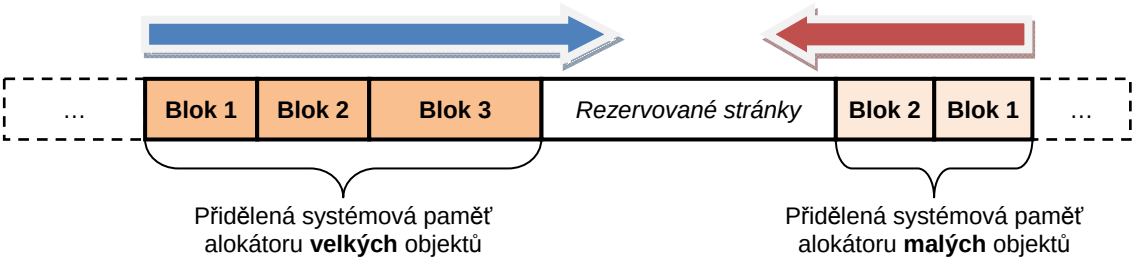
Po analýze jsem se rozhodl problém rozdělit na dvě části – alokace velkých objektů a alokace malých objektů. Pro alokaci velkých objektů jsem použil metodu založenou na alokátoru s oddělenými seznamy. Tato metoda nicméně způsobuje velkou interní fragmentaci paměti, ale u velkých objektů tento problém není tak markantní. Při alokování mnoha malých objektů by docházelo k velkému plýtvání paměti, a proto jsem se rozhodl k tomuto účelu implementovat slab alokátor, který do značné míry odstraňuje interní fragmentaci a zároveň je velmi rychlý.

Poslední z podmínek vytvoření efektivní alokace ve vícevláknových aplikacích je vyhnout se co nejvíce synchronizaci. K tomu nám dobře poslouží vyrovnávací paměti, vytvořené pro každé vlákno. V takto vytvořených vyrovnávacích pamětech je předem rezervováno několik objektů dané třídy a při požadavku o přidělení se algoritmus nejdříve pokouší objekt získat z této paměti. Jelikož je izolovaná od ostatních vláken, není potřeba synchronizace a tudíž může alokace proběhnout velmi rychle. Protože má smysl rezervovat jen objekty malé velikosti, stačí vytvořit vyrovnávací paměti na úrovni vláken pouze ve slab alokátoru. V této kapitole popíši hlavní rysy implementace jednotlivých vrstev alokátoru. Tento návrh byl částečně inspirován projektem [28].

6.1 Centrální správa paměti

Na nejnižší úrovni je vytvořena správa systémové paměti. Ta má za úkol odstínit implementační závislosti na použitém systému a zároveň slouží k jednoduché správě alokovaných úseků virtuální paměti. Jak jsem popsal v kapitole 3, tato část je vhodná k implementaci principu cache, která má za úkol snížit komunikaci s jádrem systému.

Na systému Windows a Linux jsem tuto správu implementoval odlišně, jelikož oba systémy přistupují k filosofii správy virtuální paměti jinak. U systému Windows se standardně rezervuje určitá oblast virtuální paměti. K tomu slouží parametr `MEM_RESERVE` funkce *VirtualAlloc()*. Pokud chceme zpřístupnit z této oblasti nějaký úsek, musíme ho z takto rezervované virtuální paměti namapovat na paměť fyzickou. K tomu nám opět slouží funkce *VirtualAlloc()*, tentokrát s parametrem `MEM_COMMIT`. Funkce *VirtualAlloc()* lze sice použít stejně jako *mmap()* u systému Linux, ale má jedno velké omezení. Netuším, jestli je to nějaká chyba nebo je to úmyslné, ale fakt, že tato funkce po určitém předem daném počtu volání již nealokuje žádnou paměť, není nikde dokumentovaný. Tento problém jsem popsal také v příloze A. Aby správce takto rezervované paměti nebyl složitý, pro správce velkých objektů volám commit zleva doprava a pro alokátor malých souborů zprava doleva. Díky tomu se pak musí paměť postupně vracet systému od posledního přiděleného bloku. Následující obrázek 6.1 znázorňuje, jak vypadá rozdělení virtuální paměti u systému Windows:



Obrázek 6.1: Rozdělení virtuální paměti mezi alokátory u systému Windows

U systému Linux nemáme taková omezení a dokonce lze pomocí funkce *mremap()* i rozšiřovat. Díky tomu, že se tak děje nad virtuální pamětí, je tato operace vytvořena poměrně efektivně, jelikož pouze přemapuje fyzické rámce na jiné virtuální adresy a nedochází ke skutečnému přesunu dat. Protože ale rozšíření virtuální paměti nezaručuje zachování bázové adresy, nelze tuto funkci použít. Proto se alokuje vždy nový blok a udržuje se v lineárním seznamu do té doby, než je opět uvolněn.

6.1.1 Navracení systémové paměti

Jakmile některý z alokátorů již poslední blok nevyžaduje, zavolá správce systémové paměti, aby ho uvolnil. Ten to ale okamžitě neprovede a vyčkává, dokud se využití paměti předposledního bloku nedostane na jeho polovinu. Zároveň se měří doba, po kterou se blok k uvolnění nevyužíval. Jakmile jsou splněny oba předpoklady, tak se paměť vrátí systému. Díky tomu se zabrání opakovanému volání systému, pokud se obsazení paměti pohybuje na hranici obsazení předposledního bloku.

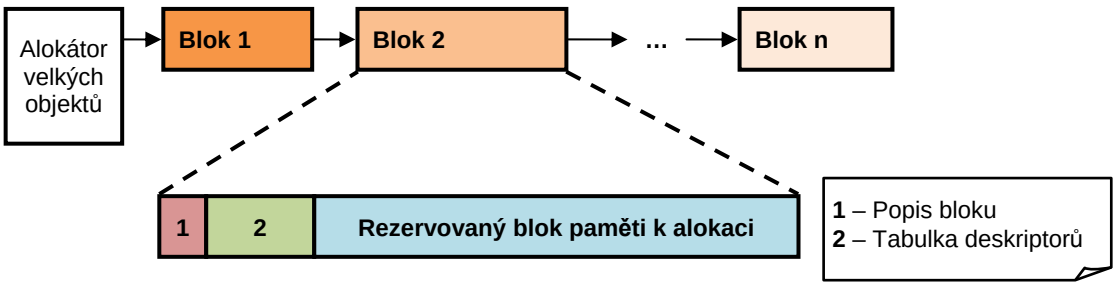
Protože se uvolňuje vždy poslední blok, nadřazené alokátory musí zajistit, aby se tento blok uvolnil co nejdříve. Tohoto výsledku jsem dosáhl tak, že při vracení paměti se volný úsek ležící v posledním bloku vloží vždy na konec seznamu volných úseků. Pokud tento úsek leží v jiném bloku, je vložen na začátek seznamu. Díky tomu jsou úseky ležící v posledním bloku přidělovány méně. Jakmile se sníží potřeba paměti, je tento blok zcela volný a může se nastartovat proces popsaný v předchozím odstavci.

6.2 Alokátor velkých objektů

Tento alokátor slouží k alokování objektů, které nemohou být uspokojeny Slab alokátorem. Jelikož Slab alokátor vytváří cache podle aktuální potřeby, nedá se přesně říct, pro které objekty se alokátor velkých objektů použije. Na počátku je konfigurace nastavena tak, aby velikost velkého objektu odpovídala velikosti větší než 16 KiB. Obecně je granularita tohoto alokátoru 4 KiB. To odpovídá velikosti jedné stránky. Dále popíši jednotlivé části tohoto alokátoru a jejich funkci.

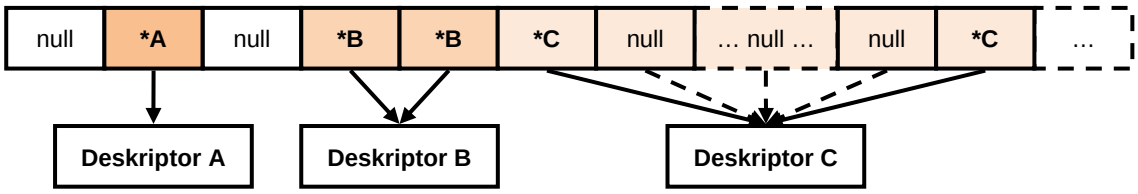
6.2.1 Logické schéma alokátoru

Na nejnižší úrovni se udržují bloky paměti získané z centrálního správce paměti. Na začátku každého bloku jsou informace, které vyžaduje centrální správa paměti: velikost bloku, zaplnění bloku, čas od posledního vyprázdnění bloku, atd. Za těmito povinnými informacemi následují data důležitá pro vlastní alokátor. První z nich je tabulka ukazatelů na deskriptor. Deskriptory slouží k popisu alokovaných úseků paměti z odpovídajícího bloku. Počet ukazatelů tabulky odpovídá počtu stránek, které obsahuje daný blok. Nejdříve jsem uvažoval o globální tabulce, která by popsala celý 32 bitový adresový prostor, ale díky tomu, že by zabrala celé 4 MiB, tak jsem od toho upustil. Navíc by tato tabulka byla velice řídká. U 64 bitového prostoru by byl tento problém neřešitelný a musela by se stejně použít víceúrovňová tabulka. Za touto tabulkou následuje vlastní blok paměti k alokaci. Obrázek 6.2 ukazuje, jak jsou tyto bloky v paměti udržovány. Na začátku je vytvořen pouze jediný blok o konstantní velikosti, která se dá změnit. Při nedostatku paměti se vytvoří další blok o stejné velikosti nebo větší, pokud se požadovaný úsek do tohoto bloku nevejde.



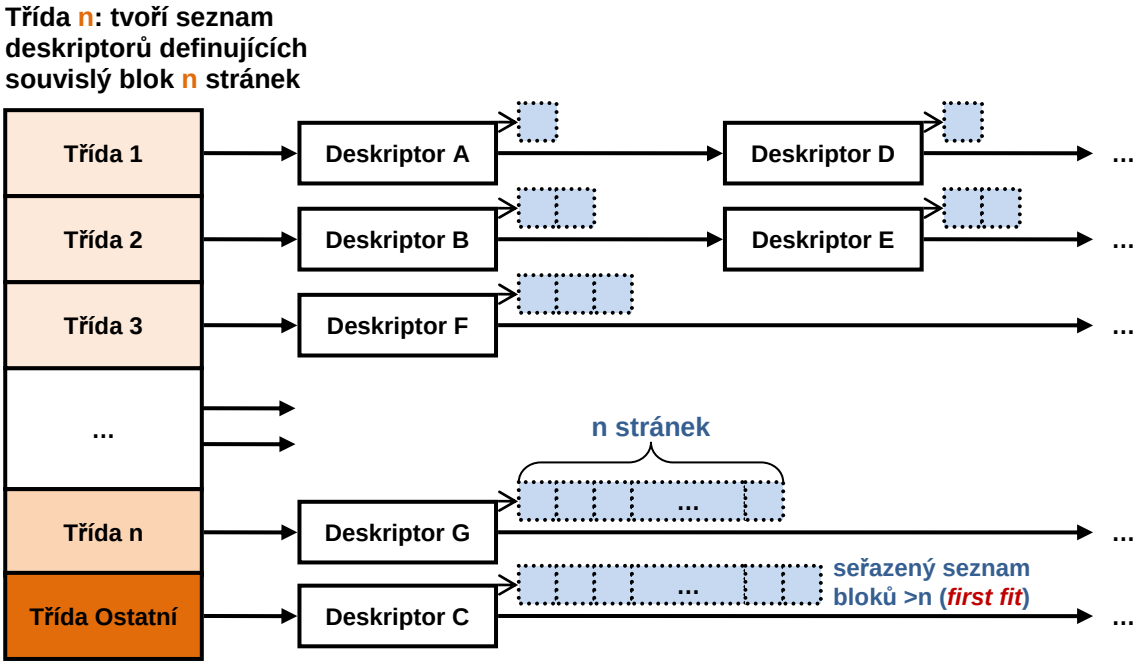
Obrázek 6.2: Struktura rezervovaných bloků paměti k alokaci

Tabulka stránek obsahuje ukazatele na deskriptory úseků paměti. Obrázek 6.3 tento pohled logicky zobrazuje. Pokud se zamyslíme nad problémem alokace, tak potřebujeme znát pouze odkaz na deskriptor z počáteční a koncové stránky souvislého bloku. Takto jsme schopni nalézt ke každé alokované adrese příslušný deskriptor a zároveň i deskriptory sousedních bloků. U alokace větších bloků paměti tak můžeme ušetřit několik cyklů, potřebných k nastavení ukazatelů na všechny stránky. Deskriptor samotný popisuje jak je daný blok velký, jestli je volný nebo přidělený a další různé příznaky.



Obrázek 6.3: Tabulka deskriptorů správce paměti bloku

Alokátor je tvořen seznamem volných bloků paměti o dané velikosti. Jelikož je granularita tohoto alokátoru stanovena na velikost stránky (4 KiB), jsou skupiny volných bloků rozděleny do tříd, které vyjadřují počet volných stránek. Počet těchto tříd není dobré stanovit neomezeně, aby nalezení volného bloku netrvalo příliš dlouho a nedegradovali jsme tento algoritmus na jiný druh lineárního prohledávání. Podrobněji si tento problém vysvětlíme v následující podkapitole. Proto jsem tento počet pevně stanovil při inicializaci na 384 tříd. Díky tomu lze v nejvyšší třídě udržovat volné bloky o velikosti 1,5 MiB, což pokryje běžnou potřebu aplikací při zachování rychlosti. Aby alokátor nebyl takto uměle omezován pouze na tyto třídy velikostí, poslední třída vyjadřuje všechny zbývající třídy (viz obrázek 6.4). Na tuto poslední třídu jsem se rozhodl použít algoritmus first fit, jelikož se snaží alokovat paměť ze začátku bloku a nedochází tak k velkému rozptylu alokovaných objektů. Výhodou je také respektování lokality odkazů a zároveň se při hledání volného bloku nemusí procházet celý seznam a nedochází tak k velké ztrátě rychlosti. Problém hromadění malých objektů na začátku bloku řeší předešlé třídy, do kterých jsou zařazeny a mohou být rychle přeskočeny/vyhledány.



Obrázek 6.4: Seznamy volných bloků zařazených do příslušných tříd

6.2.2 Způsob přidělení a uvolnění paměti

Proces alokace probíhá tak, že se požadovaná velikost zarovná na velikost stránky a poté se stanoví příslušná třída. Nejprve se zjistí, zda je v dané třídě volný úsek paměti a pokud ne, tak se postupně prohledávají vyšší třídy. Pokud je volný úsek nalezen ve vyšší třídě, rozdělí se na část, která je požadovaná a druhá je vložena podle její velikosti do odpovídající třídy. Pokud se nenalezne žádný volný blok, musí se přistoupit k alokování paměti z centrálního správce.

Při uvolňování paměti se postupuje obdobně jako u algoritmů sequential fit. Po uvolnění se vždy musí zkontrolovat sousední bloky paměti a spojit všechny volné do jednoho. Tento nově vytvořený blok je posléze vložen do odpovídající třídy. Pokud je vkládána do poslední třídy, kde se využívá algoritmu first fit, musí se vložit tak, aby výsledný seznam byl správně seřazen podle adresy.

6.2.3 Vyhledání volného úseku

Standardně lze proces vyhledání volného úseku naprogramovat jako sekvenční vyhledání prvního neprázdného seznamu. Vyhledávání začíná od třídy, která nejlépe vyhovuje požadované velikosti. Následující úsek kódu tento způsob ukazuje:

```
for (dwNewClass = dwClass; dwNewClass < CLASS_COUNT; ++dwNewClass) {
     if (!ListEmpty(&memHeap.freeBlocks[dwNewClass])) {
         pMemBlock = ListEntry(memHeap.freeBlocks[dwNewClass]);
         break;
     }
}
```

Pokud se volný úsek nalezne v poslední třídě, která odpovídá záznamu first fit a neodpovídá požadované velikosti, musí se v této třídě prohledat i ostatní úseky. Díky tomu má tento algoritmus obecně lineární časovou složitost. Protože se ale v praxi nealokují často příliš velké objekty, vykazuje tento algoritmus konstantní složitost, protože se velmi často volný úsek nalezne v nižší třídě nebo první prvek u záznamu first fit dostačuje pro uspokojení požadavku.

Nicméně pokud je tříd příliš mnoho a pokud se rovnoměrně nezaplní volnými úseky, což se může stát, když se aplikace pokusí alokovat mnoho objektů, které nebude okamžitě navracet, může dojít k neustálému procházení všech tříd, než se volný úsek nalezne až v poslední. Tento problém nejlépe ukazuje příklad, kdy se opakovaně alokuje a uvolňuje objekt stejné třídy. Pokud odpovídá první třídě, pak se musí neustále procházet všechny třídy. V našem případě 384, což je již poměrně velký cyklus. Proto jsem vytvořil algoritmus založený na bitovém poli, který tento cyklus snížil na minimum a je mnohem odolnější proti těmto extrémům.

Každé třídě jsem přidělil jeden bit. Pokud je tento bit nastaven, existuje v této třídě alespoň jeden volný úsek, v opačném případě je seznam prázdný. 384 tříd odpovídá poli o velikosti 48 bajtů. U procesorů Intel existují řadu let instrukce MMX, které mají šířku slova 64 bitů. Pokud bychom uvažovali instrukce SSE, tak lze pracovat až se 128 bity, ale bohužel základní verze neobsahuje instrukce pro práci s celým takto velkým slovem.

Omezil jsem se proto na 64 bitů, které jsou také dostačující. Samotné vyhledávání se dá přirovnat k paralelnímu algoritmu hledání na stromu. Algoritmus hledající první nenulový bit v poli vypadá následovně:

```
UInt32   dwIndex = 0;
UInt64*  pBlock1 = (UInt64*)&dbBitField;
for (int i = 0;  i < CLASS_COUNT/64; ++i, dwIndex += 64, pBlock1++) {
     if (pBlock1[i] != 0) break;
}

UInt32*  pBlock2 = (UInt32*)pBlock1;
if (pBlock2[0] == 0) {
    pBlock2 += 1; dwIndex += 32;
}

dwIndex += BitScanForward(pBlock2);
```

Tento kód je nezávislý na použitém procesoru a umožňuje snížit maximální počet cyklů z 384 na 6 případně na 7, pokud počítáme i závěrečný podmíněný příkaz, který se také překládá na skok. V posledním kroku je použita funkce pro nalezení indexu prvního nenulového bitu. Na procesorech Intel lze k tomuto účelu použít instrukci `BSR` a tím zcela odstranit cyklus. Pokud na dané platformě obdobná instrukce není, je nutné tuto funkci nahradit cyklem, který se dá také optimalizovat.

Abychom tento algoritmus rozšířili na hledání bitu od určité pozice, musíme nejdříve vypočítat pozici prvního 64 bitového bloku a v něm ještě pomocí masky vytvořené pomocí operace posunu odfiltrovat nepotřebné bity z bitového pole. Tento kód vypadá následovně:

```
UInt32   i = dwClass/64;
UInt32   dwIndex = i * 64;
pBlock1[i] &= 0xFFFFFFFFFFFFFFFF << (dwClass - dwIndex);
```

6.2.4 Deskriptory úseků paměti

Jak jsem již psal, na začátku rezervovaného bloku se nachází tabulka ukazatelů na deskriptory. Tyto deskriptory se musí také někde alokovat, a protože mají konstantní velikost, přišlo mi jako nejlepší způsob použít Slab cache, která se vytvoří přesně pro velikost těchto deskriptorů.

Na začátku, kdy se rezervuje první blok, se vytvoří úsek, který odpovídá celému volnému bloku. Tento úsek je vložen do poslední třídy a při první alokaci je rozdělen. Pokaždé, když je nějaký úsek přidáván nebo odebírán, se musí správně nastavit bitové pole, které využívá algoritmus pro hledání volných úseků. K tomuto účelu jsem vytvořil dvě makra:

```
SET_BIT(field,i) (((char*)(field))[((i) >> 3)] |=  (1 << ((i) & 7)))
CLR_BIT(field,i) (((char*)(field))[((i) >> 3)] &= ~(1 << ((i) & 7)))
```

Pokud se volný úsek vkládá do nejvyšší třídy, která odpovídá algoritmu fist fit, musí se vložit tak, aby byly všechny úseky seřazeny podle své adresy. Aby se nemusela pokaždé sekvenčně procházet část seznamu nebo dokonce celý seznam, vytvořil jsem u některých operací způsob, jak tento průchod omezit. Předpokladem ale je, že se při těchto operacích pracuje s úseky, kde alespoň

jeden leží v seřazeném seznamu z nejvyšší třídy. Pokud u operace spojování volných bloků *Merge()* existuje pouze jeden úsek ležící v nejvyšší třídě, tak se z tohoto seznamu neodstraní a pouze se rozšíří tak aby odpovídal velikosti spojených úseků a pozici úseku s nejnižší adresou. Zbývající úseky se odstraní. Pokud u této operace leží více úseku v nejvyšší třídě, je použit ten, který leží na nejnižší adrese a opět se provede pouze změna velikosti a odstranění zbývajících úseků. Operace rozdělení volného bloku *Split()* je o něco jednodušší. Pokud se úsek k rozdělení nachází v nejvyšší třídě a pokud se po rozdělení zbývající úsek nachází opět v nejvyšší třídě, jeho pozice se v tomto seznamu nezmění. Jestli jsou u operací použity úseky, kde všechny leží mimo nejvyšší třídu a jejich výsledkem je vytvoření úseku patřícího do nejvyšší třídy, musí se provést sekvenční vložení.

6.3 Alokátor malých objektů (Slab alokátor)

Při inicializaci je nastaveno, aby tento alokátor přiděloval objekty do velikosti 16 KiB. V úvodu jsem již avizoval, že pro tento účel použiji Slab alokátor, který je velice rychlý a umožňuje snížit interní fragmentaci. Slab alokátor vyžaduje ke své činnosti ještě subalokátor k získání souvislého úseku stránek. Rozhodoval jsem se mezi použitím alokátoru velkých objektů, který jsem vytvořil a Buddy systémem používaným v jádrech operačních systémů. Nakonec jsem se rozhodl pro Buddy systém, který je o něco rychlejší a zároveň odpadne problém s cyklem, který by mohl nastávat u alokátoru velkých objektů, který pro alokaci deskriptorů používá Slab alokátor. Navíc by malé úseky paměti pro Slab alokátor dlouhodobě rozdělovaly paměť určenou pro velké objekty a tím by mohlo docházet k externí fragmentaci. Nejprve tedy popíši implementaci Buddy systému.

6.3.1 Buddy systém

Obdobně jako u alokátoru velkých objektů se udržují bloky paměti získané z centrálního správce paměti. Nejlépe tuto skutečnost ilustruje obrázek 6.3. Jediným hlavním rozdílem je, že se místo tabulky ukazatelů na deskriptory používá tabulka informací o stránkách. Aby jednotlivé prvky nezabíraly moc místa, použil jsem pro atributy bitová pole, ale i tak zabírá více jak prostý ukazatel. Výhodou tohoto řešení je, že již není potřeba žádný dodatečný subalokátor a nemůže vzniknout žádný cyklus při nedostatku paměti. Navíc se nerezervují tak velké bloky, takže těchto prvků není příliš mnoho a tato režie je přijatelná.

Tento alokátor umožňuje přidělovat pouze stránky v počtu 2^n. Proto se jako parametr alokace nepředává skutečná velikost v bajtech, ale pouze řád n (*order*). Jelikož nemá smysl přidělovat všechny možné řády n, stanovil jsem maximum 4. To odpovídá přidělení úseku o maximální velikosti 64 KiB, což je v našem případě dostačující a uplatní se pouze u největších Slab cache. Díky tomuto omezení musí algoritmus alokace a uvolnění vykonat maximálně 4 cykly a proto je tak rychlý. Aby mohl tento algoritmus pracovat správně, musel jsem v inicializaci provést rozdělení celého bloku na úseky o velikosti 64 KiB a ty vložit do příslušného seznamu volných bloků.

Samotný proces alokace na začátku probíhá velice podobně jako u alokátoru velkých objektů. Nejprve se najde nejbližší řád, ve kterém je nějaký volný úsek. Pokud je tento řád vyšší než požadovaný, musí se provést rozdělení tohoto úseku. Celý úsek se dělí neustále na poloviny, do té doby než se získá úsek požadované velikosti, který se vrátí, a ostatní úseky jsou vložený do seznamů volných úseků daného řádu. Nejlépe to ilustruje obrázek 5.6. Samotný algoritmus vypadá následovně:

```
dwSize = 1 << dwCurrOrder;          // dwCurrOrder – nejbližší volný úsek
while (dwCurrOrder > dwOrder) {     // dwOrder – požadovaná velikost
       dwCurrOrder--;
       iBuddy = iPage + (dwSize >>= 1);
       BuddyInsert(iBuddy,dwCurrOrder);
}
```

Při vložení volného úseku do příslušného seznamu se musí zaznamenat mezi informace vedené v popisu stránky, na které začíná, také jeho velikost v podobě čísla řádu. Bez této informace by nebylo možné správně spojovat volné přilehlé dvojice (*buddy*). Uvolňování popisuje následující algoritmus:

```
iPage = dwBase/PAGE_SIZE;             // první stránka uvolňovaného úseku
for (; dwOrder < MMB_MAX_ORDER; ++dwOrder) {
     iBuddy = iPage ^ (1 << dwOrder); // výpočet přilehlého úseku

     // pokud není úsek volný a nebo stejného řádu je spojování přerušeno
     if (!pageList[iBuddy].bFree ||
          pageList[iBuddy].dwOrder != dwOrder) break;

     pPage = &pageList[iBuddy];
     ListDel(&pPage->list);   // přilehlý úsek je odstraněn
     iPage &= iBuddy;         // po spojení se vybere nejlevnější stránka
}
MemBuddyInsert(iPage,dwOrder);
```

Velkou výhodou tohoto algoritmu je, že používá pouze bitové operace a díky omezení cyklu je nutné vykonat relativně málo rychlých instrukcí.

6.3.2 Vytvoření Slab cache

Malý problém nastává hned u inicializace Slab alokátoru. K tomu aby mohla být vytvořena nějaká nová Slab cache, musí se někde alokovat deskriptor popisující tuto strukturu. A k tomu účelu slouží opět jiná Slab cache. Pokud bychom tuto cache vytvářeli standardně, došlo by k nekonečné rekurzi a nebylo by to řešitelné. Proto existuje na globální úrovni jediná Slab cache, nazvaná cacheCache, která slouží k tomuto účelu a jako jediná nemá deskriptor uložený přímo v této cache.

Dále vzniká problém s deskriptorem Slabu, který se alokuje mimo úsek paměti s objekty (OFF_SLAB). Podrobnosti lze dohledat v kapitole 5. Díky tomu, že oproti deskriptorům Slab cache nemají deskriptory Slabu konstantní velikost, musí se při inicializaci vytvořit obecné Slab cache, které jsou připraveny pro různé velikosti. Tyto Slab cache jsou nazvány cacheSizes. Protože i tyto cache mohou obsahovat OFF_SLAB, musí se tyto Slab cache vytvářet od nejmenší k největší a

zároveň musí menší Slab cache uspokojit požadavky na alokaci větších cache. Nejmenší cache tedy nemůže používat OFF_SLAB.

Pokud odhlédnu od inicializace, která má určitá úskalí, popíši hlavní kroky, které jsou nutné k vytvoření Slab cache. Nejdůležitějšími informacemi k vytvoření cache jsou velikost objektu a jeho zarovnání v paměti. Pokud není vyžadováno žádné zarovnání, je standardně nastaveno zarovnání na velikost slova, aby byla práce s takto alokovaným blokem efektivní. Podle požadované velikosti objektu, patřičně zarovnané, se určí, jestli je deskriptor Slabu umístěn mimo (OFF_SLAB) nebo je součástí rezervovaného úseku s objekty. U větších objektů je tento správce umisťován mimo a očekává se, že dojde k rozmístění více objektů do rezervovaného úseku paměti. Dále se určuje počet stránek, které se musí alokovat pomocí Buddy systému, tak aby je co nejlépe vyplnily objekty požadované velikosti a nevznikala velká interní fragmentace. Pokud po tomto určení velikosti zbývá prostor pro deskriptor Slab, který jsme předtím označili jako OFF_SLAB, tak se umístí do tohoto úseku. Zbývající režie, která vznikla nezaplněním celého úseku objekty, se použije k barvení cache.
Barvení cache. O tomto problému jsem více psal v kapitole 3. V inicializaci je pouze vypočítáno počet barev a nastavena první jako výchozí. Pokaždé když je nutné vytvořit nový Slab, je číslo této barvy cyklicky měněno. Na závěr se podle velikosti alokovaného objektu určí maximální počet objektů udržovaných ve vyrovnávacích pamětech každého vlákna a také v jakých dávkách jsou tyto vyrovnávací paměti plněny, pokud jsou prázdné.

6.3.3 Přidělení a uvolnění paměti

Nyní popíši, jak probíhá alokace paměti ze Slab cache bez použití vyrovnávacích pamětí u vláken. Nejdříve se zjistí, jestli některý Slab ze seznamů slabsPartial a slabsFree, v tomto pořadí, neobsahuje volné objekty. Pokud žádný takový neexistuje, je volána funkce *SlabCacheGrow()*, která vytvoří Slab podle popisu deskriptoru Slab cache. Tato funkce získá potřebný úsek paměti pomocí Buddy alokátoru a případně zvlášť alokuje deskriptor Slab, pokud je označen jako OFF_SLAB. Dále určí následující barvu a provede inicializaci správce Slabu (viz obrázek 5.3). Takto vytvořený Slab je posléze vložen do seznamu slabsFree a je znovu proveden pokus o alokaci objektu.

Samotné získání objektu probíhá velice podobně jako u optimálního alokátoru, který jsem popsal v kapitole 5. Následující kód tento algoritmus ukazuje:

```
// pSlab->pBase – bázová adresa úseku alokovaného pro objekty
// pSlab->free  – index ukazující na první prvek volného objektu
// pSlab->dwManager – pole indexů viz obrázek 5.3
char*  pObj = pSlab->pBase + pSlab->free*pCache->dwObjSize;
pSlab->free = pSlab->dwManager[pSlab->free];
```

Jakmile je ze Slabu alokován jediný objekt, okamžitě se přesouvá do seznamu slabsPartial. Uvolnění paměti probíhá přesně opačně:

```
int dwObj = (pObj-pSlab->pBase) / pCache->dwObjSize;
pSlab->dwManager[dwObj] = pSlab->free;
pSlab->free = dwObj;
```

6.3.4 Použití vyrovnávacích pamětí

Při vytváření Slab cache se určí výchozí hodnoty, které udávají maximální počet předalokovaných objektů ve vyrovnávací paměti jednotlivých vláken. Dále je každé Slab cache přiřazen jedinečný klíč, pomocí kterého lze přistupovat k lokálním datům vlákna. K vygenerování tohoto klíče nám operační systémy poskytují příslušné funkce. Na systému Windows je to funkce *TlsAlloc()*. Systém Linux poskytuje funkci *pthread_key_create()*, která navíc umožňuje specifikovat callback funkci, která se volá při ukončení vlákna jako destruktor objektu asociovaný s tímto klíčem.

Jelikož individuální vlákna nemusí vyžadovat alokaci objektů všech velikostí, které poskytují jednotlivé Slab cache, není dobré při inicializaci vlákna okamžitě vytvářet všechny vyrovnávací paměti. Navíc u systému Linux je velice obtížné zachytit moment, kdy je některé vlákno vytvořeno. Proto jsem samotné vytvoření vyrovnávací paměti přesunul do funkce určené k alokování objektů z dané Slab cache. Nejdříve se tato funkce pokusí získat data asociovaná s klíčem vygenerovaným pro konkrétní Slab cache. U systému Windows se používá funkce *TlsGetValue()* a u systému Linux *pthread_getspecific()*. Pokud s tímto klíčem nebyla doposud asociovaná žádná hodnota, obě funkce vrací NULL. Díky tomu lze zjistit, že je nutné vytvořit vyrovnávací paměť.

Vyrovnávací paměť je ve skutečnosti pouze pole ukazatelů, které se chová jako zásobník. Tato paměť má následující strukturu:

```
typedef struct {
    int     dwAvail;    // počet objektů na zásobníku (vrchol zásobníku)
    int     dwLimit;    // velikost zásobníku
    int     dwBatch;    // počet objektů k naplnění při prázdném zásobníku
    void*   pStack[];   // pole ukazatelů o velikosti dwLimit
} THREAD_CACHE;
```

V případě že není tato vyrovnávací paměť na daném vláknu asociována s klíčem Slab cache, je vytvořena a pomocí systémových funkcí s tímto klíčem asociována. Na systému Windows je to funkce *TlsSetValue()* a na systému Linux *pthread_setspecific()*.

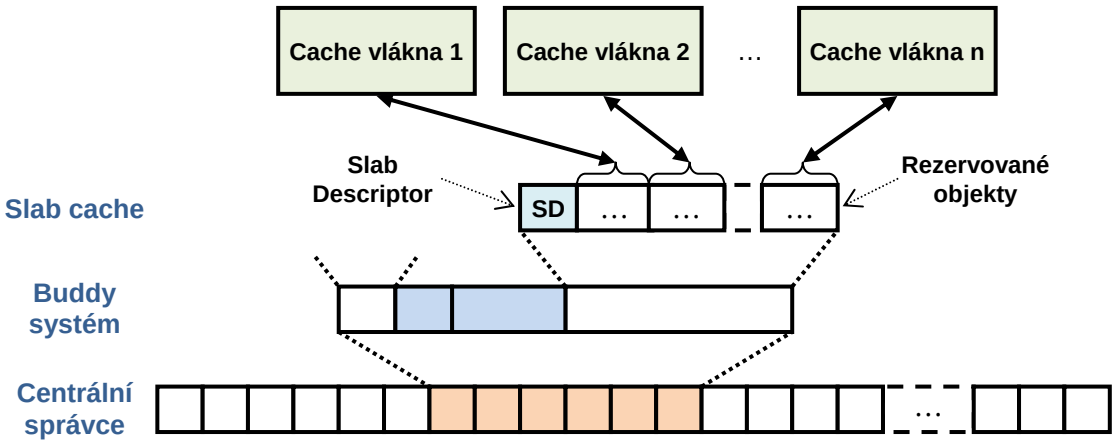
Přidělení a uvolnění paměti s použitím vyrovnávacích pamětí na každém vláknu popisuje následující kód:

```
void* SlabObjAlloc(SLAB_CACHE* pCache) {
    THREAD_CACHE* pTC = TC_DATA(pCache);    // získání vyrovnávací paměti
    if (pTC->dwAvail) {
        return pTC->pStack[--pTC->dwAvail]; // alokace z vrcholu TC
    } else {
        return SlabTCRefill(pCache);        // naplnění TC ze Slab cache
    }
}

void SlabObjFree(SLAB_CACHE* pCache, void* pObj) {
    THREAD_CACHE* pTC = TC_DATA(pCache);    // získání vyrovnávací paměti
    if (pTC->dwAvail >= pTC->dwLimit) {
        SlabTCFlush(pCache,pTC);            // návrat objeků do Slab cache
    }
    pTC->pStack[pTC->dwAvail++] = pObj;     // návrat na vrchol zásobníku
}
```

Tento způsob asociace vyrovnávací paměti s vláknem funguje bezchybně do té doby, než vytvoříme příliš mnoho klíčů. Např. u systému Windows lze udržovat maximálně 1088 takových klíčů. U starších verzí to bylo dokonce pouze 80 případně 64. Zároveň by nastával problém, pokud i cílová aplikace používá toto lokální úložiště a alokátor by ho mohl zcela zabrat pro sebe. Navíc u systému Linux vzniká problém s detekcí ukončení vlákna. To jsem vyřešil tak, že jsem při vytvoření klíče pro lokální data vlákna přidal i destruktor, který uvolní veškerou paměť asociovanou s tímto vláknem. Nicméně nastal problém při implementaci kontroly úniku paměti, která se spouští při každém ukončení vlákna. Nejdříve se totiž musí vyprázdnit všechny vyrovnávací paměti, než je tato kontrola volána. Protože jsou tyto paměti uvolňovány postupně pomocí asociovaného destruktoru, je obtížné určit, který destruktor bude volaný jako poslední. K tomu by mohlo posloužit nějaké počítadlo, vytvořené jako další lokální data vlákna, ale vedlo by to k dalšímu plýtvání tohoto prostoru. Proto jsem tento problém vyřešil jinak. Celý alokátor si asociuje pouze jediný klíč, který použije pro uložení vektoru se všemi vyrovnávacími pamětmi pro dané vlákno. S tímto vektorem se pak bude pracovat stejně, jak jsem popsal. Jediný rozdíl bude v tom, že klíčem asociovaným se Slab cache bude index do tohoto pole. Pokud bude vlákno ukončeno, vyvolá se pouze jediný destruktor a po uvolnění všech vyrovnávacích pamětí se již může volat kontrola úniku paměti.

Na obrázku 6.5 je znázorněna celá hierarchie Slab alokátoru. Obrázek může navozovat dojem, že rezervované objekty ve vyrovnávací paměti musí ležet vedle sebe. Tomu tak ve skutečnosti obvykle není a může se i stát, že se objekty nachází v různých Slabech.



Obrázek 6.5: Vztah centrální paměti, slab cache a vyrovnávací paměti vlákna

6.4 Implementace funkce malloc

Tato kapitola má za úkol zastřešit doposud popsané části celého správce paměti a spojit je v jeden celek. Zároveň popisuje, jak lze vytvořit statickou knihovnu na systémech Windows a Linux, tak aby nahradila standardní knihovnu jazyka C, poskytující správu paměti.

6.4.1 Inicializace a uvolnění alokátorů

Aby mohl alokátor správně fungovat, je nutné inicializovat všechny jeho vnitřní struktury před prvním voláním funkce malloc. Zároveň je nutné uvolnit všechny prostředky při ukončování programu. Pokud program pracuje s vlákny, potřebujeme také zjistit okamžik spuštění a ukončení vlákna. Systémy Windows a Linux jsou ve způsobu řešení tohoto problému značně odlišné a proto implementační detaily na obou systémech popíši zvlášť.

6.4.1.1 Implementace na systému Windows

Dynamické knihovny na systému Windows mají možnost inicializační rutiny, popřípadě rutiny k uvolňování rezervovaných prostředků umístit do standardní funkce *DllMain()*. Ta umožňuje zachytit události o vytvoření procesu/vlákna a také o jejich ukončení.

Jelikož jsem se rozhodl implementovat alokátor pomocí statické knihovny, která takovou funkci standardně nepodporuje, musel jsem použít jeden nedokumentovaný trik, o kterém se dá dočíst na internetu např. u „boost“ knihovny. Využívá se znalosti linkeru a způsobu, jakým standardní kompilátor systému Windows vytváří inicializační rutiny. Nejprve si ukažme tento princip na krátkém kódu:

```
static void NTAPI OnTlsCallback(HINSTANCE, DWORD, PVOID);
#pragma data_seg(".CRT$XLB")
    _TLSCB pfThreadCallback = OnTlsCallback;
#pragma data_seg()
```

Tento kód zajistí umístění ukazatele na funkci *OnTlsCallback()* do segmentu pojmenovaného jako `".CRT$XLB"`. Toto jméno bylo získáno z interních struktur kompilátoru, které říká linkeru, jak mají být rozmístěny vektory ukazatelů v adresáři TLS (**T**hread **L**ocal **S**torage). Adresář TLS je obsažen ve spustitelném PE formátu a slouží mimo jiné ke specifikování lokálních dat pro každé vlákno v době překladu. Konkrétně segment `".CRT$XLB"` je určen pro vytvoření tzv. vektoru funkcí *TLS-Callback*. O vektoru mluvím záměrně, protože těchto ukazatelů na funkce zde může být definováno více. Díky tomu, že linker spojuje stejně pojmenované segmenty do jednoho úseku paměti, lze tyto callback funkce definovat v různých knihovnách. Na tento vektor se lze za běhu odkazovat pomocí ukazatelů `__xl_a` (začátek vektoru) a and `__xl_z` (konec vektoru).

Samotná funkce *OnTlsCallback()* je definovaná úplně stejně jako funkce *DllMain()* a pomocí typu volání (`DLL_THREAD_ATTACH`, `DLL_THREAD_DETACH`, …) lze volat příslušné inicializační a uvolňovací rutiny.

6.4.1.2 Implementace na systému Linux

Systém Linux neposkytuje obdobnou funkci jako systém Windows, ale naštěstí je tento problém možné řešit jinak. Inicializace alokátoru lze provést pomocí funkcí s atributem konstruktoru a destruktoru. Tyto funkce se volají před samotným spuštěním funkce main a po jejím ukončení.

Nicméně mohou nastat problémy, pokud se používá funkce malloc u inicializace globálních objektů. Obecně funkce konstruktoru a destruktoru negarantují pořadí jejich spouštění a to může způsobovat problémy. Proto je nejlepším řešením u obou systémů provádět inicializaci u prvního volání funkce malloc. U destruktoru to ale takto nelze dělat. Nicméně jsem zjistil, že funkce s atributem destruktoru je volána až po vykonání všech globálních destruktorů tříd a funkcí registrovaných pomocí standardní funkce *atexit()* a proto beru toto řešení jako dostačující. Následuje ukázka funkce s atributem:

```
void OnProcessInit() __attribute__ ((constructor));
void OnProcessTerm() __attribute__ ((destructor));
```

Událost vytvoření vlákna jsem popsal v kapitole o použití vyrovnávacích pamětí. Je řešeno ve Slab alokátoru, pokud nejsou inicializovaná lokální data vlákna. V této kapitole jsem popsal i zachycení události konce vlákna, které je řešeno pomocí asociace destruktoru s lokálním klíčem vlákna. Jak jsem zjistil, tento destruktor se nevolá na lokální data asociovaná s hlavním procesem. Ale tento problém řeší funkce *OnProcessTerm()* popsaná výše.

6.4.2 Vytvoření vektoru velikostí Slab cache

K tomu, aby mohl fungovat Slab alokátor, je potřeba vytvořit Slab cache různých velikostí. Pokud by existovalo mnoho velikostí, muselo by se při každé alokaci procházet v cyklu mnoho záznamů. Proto jsem vytvořil vektor ukazatelů na Slab cache, který umožní rychlý přístup pomocí indexu vypočítaného z požadované velikosti. Na začátku se zjistí, pro které velikosti je vhodné vytvořit Slab cache, aby nedocházelo k velké interní fragmentaci. Přijatelnou fragmentaci jsem stanovil na 20% nevyužití paměti. Velikost tohoto vektoru jsem určil 1024 prvků s krokem velikosti objektu 32 bajtů. Aby nebylo na začátku vytvořeno příliš mnoho Slab cache, tak některé indexy ukazují na stejný Slab cache a jsou tak sdíleny. Za běhu se u každého indexu udržuje statistika o fragmentaci a pokud přesáhne určitou hranici, je pro tuto velikost vytvořena nová Slab cache.

6.4.3 Nahrazení standardní funkce malloc

Aby bylo možné nahradit standardní funkci malloc nově definovanou, je nutné do projektu vložit hlavičkový soubor, který toto nahrazení jednoduše zajistí bez nutnosti další úpravy zdrojových souborů. V implementaci jsem použil následující způsob definice takového hlavičkového souboru:

```
#define malloc    malloc_userdef
extern "C"  void* malloc(size_t size);

void* operator new(size_t size);
void* operator new[](size_t size);
```

Preprocesor zajistí nahrazení všech výskytů slova malloc na nově definovaný název, který se nachází v novém alokátoru. Tato změna názvu není nezbytně nutná, ale díky ní nevznikne problém při spojování programu, kde uživatel může použít i standardní knihovnu a existovaly by dvě funkce se stejnými názvy. Takto navíc v jednom projektu můžeme použít standardní funkci malloc a nově vytvořenou. Operace new pro C++ se řeší jako standardní přetížení operátorů.

7 Nástroje pro kontrolu paměti

Pokud v dnešní době vytváříme nový alokátor paměti, je dobré do návrhu začlenit i mechanismy, které umožní snadněji odhalovat chyby, způsobené nesprávnou manipulací s pamětí. Tyto chyby jsou u jazyků C/C++ poměrně rozšířené a velice špatně se hledají a odstraňují. Pokud k tomu přičteme navíc paralelismus, který přináší i použití vláken, je tento problém často velmi komplikovaný a k lokalizování takové chyby mnohdy pomůže až náhoda.

7.1 Implementace kontroly paměti

V následujících kapitolách popíši typické chyby, které vznikají v aplikacích při manipulaci s dynamickou paměti. Zároveň je zde popsána implementace různých mechanismů jak těmto chybám předcházet, případně usnadnit jejich řešení.

7.1.1 Překročení hranice paměti

Snad nejtypičtější chybou při práci s pamětí je překročení její hranice (*buffer overflow/underflow*). Tento problém je velice nepříjemný, protože se v řadě případů projeví až za delší dobu nebo se v horším případě neprojeví nikdy. Pokud se navíc tato chyba vyskytuje u paralelní aplikace, začne se projevovat velice nedeterministicky a její odhalení je o to složitější. Navíc je tento problém zároveň spojen s bezpečnostními riziky. Týká se to ale především přetečení zásobníku, které v této práci nebudu řešit. Velmi často již nějakou podporu kontroly zásobníku poskytuje samotný překladač. U alokátoru hromady (*heap*) tento problém tak velké bezpečnostní riziko nepředstavuje, jelikož je tuto chybu mnohem obtížnější zneužít, protože je většinou těsně svázána s implementací alokátoru. Nicméně tento problém může značně ovlivnit stabilitu celé aplikace.

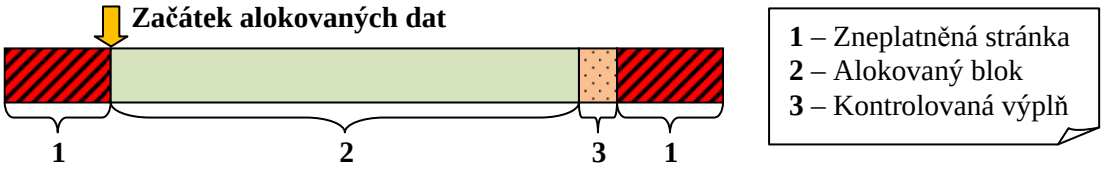
Detekce přetečení paměti se nejčastěji provádí tak, že je alokováno o něco více paměti, než je požadováno a tento vzniklý prostor se využije k „nastražení“ pasti na obou hranicích alokované paměti. Jelikož se můj alokátor skládá z více subalokátorů a každý z nich řeší tento problém trochu jinak, popíši je samostatně. Povolení této kontroly se provádí pomocí makra `OVERFLOWCHECK`.

7.1.1.1 Alokátor velkých objektů

U tohoto alokátoru lze s výhodou využít jeho vlastnost, kterou je alokace bloků s granularitou o velikosti jedné stránky. Při alokaci přičteme k požadovaným stránkám ještě dvě stránky navíc, které použijeme jako past na začátku a konci alokovaných stránek. Tato past bude spočívat v nastavení přístupových práv k těmto stránkám na úplné minimum, což nám umožňuje správce virtuální paměti cílového operačního systému. Tento způsob přináší několik výhod. Zaprvé aplikace způsobí výjimku při pokusu o zápis do této oblasti a díky tomu může debugger okamžitě odchytit přesné místo v kódu,

kde k tomuto problému došlo. A zadruhé lze tímto způsobem odchytit i pokus o čtení, což ostatní metody neumožňují. Nicméně je nutné dodat, že díky této ochraně zvedneme využití virtuální paměti. Neboť je tato kontrola prováděna pouze u aplikací přeložených k jejich ladění, tak pozitiva stále převažují nad těmito negativy.

Protože aplikace většinou požaduje paměť o velikosti, která není zarovnaná na velikost stránky, tak znepřístupnění následující stránky nevyřeší přepsání menších úseků paměti, jelikož mezi alokovaným blokem a začátkem následující stránky často vzniká mezera. Tento problém lze vyřešit tak, že se tato mezera vyplní nějakou hodnotou, která se v systému vyskytuje poměrně zřídka. Při uvolnění paměti je tato hodnota zkontrolována a pokud byla nahrazena jinou hodnotou, tak jsme detekovali přetečení. Tato metoda poměrně bezpečně detekuje přetečení paměti, ale bohužel již nedetekuje jeho příčinu, ani nelze zabránit čtení z této oblasti. Aby nebylo hledání příčiny pro programátora tak obtížné, alokátor zároveň udržuje informace o tom, komu tato paměť patří. Poté již je na schopnostech programátora, jestli tuto chybu identifikuje. Dále může být problém zvolit hodnotu, která je do takové mezery zapisována. Je jistá pravděpodobnost, že v aplikaci dojde k přetečení a tato hodnota bude náhodou přepsána úplně stejnou, díky čemuž se toto přetečení neodhalí. Proti tomuto problému se nadá příliš bránit, ale naštěstí je tato pravděpodobnost poměrně malá. Následující obrázek 7.1 znázorňuje alokovanou paměť s ochranou:



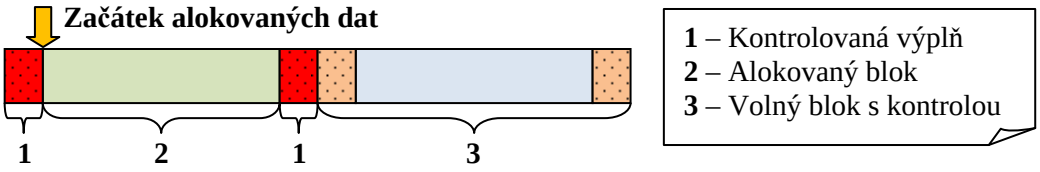
Obrázek 7.1: Detekce přetečení paměti u alokátoru velkých objektů

Při uvolňování paměti je nutné nejdříve odečíst od předaného ukazatele velikost jedné stránky, abychom se dostali na skutečný začátek alokovaného bloku. Dále se zkontroluje oblast 3 a pokud byla změněna, je vyvolána výjimka, která se pokusí přerušit běh programu a přepnout se do režimu ladění. Zároveň zobrazí vlastníka této oblasti paměti.

7.1.1.2 Alokátor malých objektů

Alokátor malých objektů nedovoluje zneplatnit celé stránky, jelikož objekty leží těsně za sebou a navíc by to způsobovalo obrovskou fragmentaci paměti. Jelikož je malých objektů alokovaných obvykle o hodně více než velkých, nebylo by dobré vytvářet příliš velké kontrolní oblasti. Proto se použijí pouze dvě slova (někdy zvané *poison words*) na začátku a konci alokovaného bloku. Opět platí, že by tato slova měla být co nejvíc jedinečná.

Protože u tohoto alokátoru leží objekty těsně vedle sebe a jsou chráněny pouze jedním slovem, tak se označují i nealokované objekty. Aby se tyto značky rozlišily, mají jinou hodnotu a označují se jako ACTIVE a INACTIVE. Obrázek 7.2 ukazuje dva objekty ležící v paměti vedle sebe.



Obrázek 7.2: Detekce přetečení paměti u alokátoru malých objektů

Kontrola překročení hranic paměti se u tohoto alokátoru provádí jak při uvolňování, tak i při přidělování paměti. Pokud dojde k přepsání oblasti v nepřiděleném bloku paměti, je poměrně složité určit místo, kde k tomu došlo. K usnadnění hledání této příčiny lze vypsat vlastníky okolních objektů, ale v mnoha případech je to neúčinné, jelikož mohlo být provedeno mnoho alokací, než k této detekci dojde. U vícevláknové aplikace je tento problém o to větší. Každopádně je tato kontrola užitečná alespoň k detekci špatného přístupu k paměti.

7.1.1.3 Další možnosti rozšíření

V této podkapitole popíši další možnost jak vylepšit kontrolu špatného přístupu k paměti, který jak jsem mnohokrát napsal, způsobuje značné problémy a špatně se detekuje. Nicméně je to zatím jen teoretická úvaha a předpokládám, že ji implementuji v dalším rozšíření projektu.

Nejúčinnějším nástrojem k této detekci a přesnému určení příčiny je bezpochyby změna přístupových práv k oblastem paměti. V mé implementaci již tuto techniku částečně používám, ale lze ji použít v mnohem větší míře. V mé praxi jsem se mnohokrát setkal s problémy, kdy špatně napsaný úsek kódu v některých situacích zcela náhodně zapisoval do určité oblasti v paměti, a to často i do vnitřních struktur alokátoru. Pokud aplikace havarovala, bylo téměř nemožné najít příčinu této chyby. Jelikož se často jednalo o vícevláknové aplikace, tak i když byla k dispozici všechna vstupní data, stejnou chybu se již nepodařilo vyvolat.

Díky těmto problémům mě napadlo, že by bylo ideální udržovat přístupné pouze úseky paměti nezbytné pro chod aktuální funkce. Myšlenka je taková, že by se po inicializaci správce paměti uzamknula veškerá virtuální paměť dostupná aplikaci. Teoreticky by se dalo uzamknout i zásobník nebo datovou sekci aplikace, ale muselo by se to provádět velice opatrně. Kódová sekce aplikace již bývá většinou standardně nastavena pouze pro čtení a není ji potřeba dále chránit. Dále by na začátku každé funkce pracující s dynamickou pamětí muselo existovat volání funkce *UnlockMemory*, které by zpřístupnilo tyto oblasti paměti. A na konci funkce by se paměť opět vrátila pomocí *LockMemory*. Tyto zámky by se musely vytvořit s počítadlem, aby nedošlo ke zneplatnění paměti v některém podprogramu. Pokud bychom chtěli tento princip použít, s velkou pravděpodobností by byla nezbytná spoluúčast programátora, který by tyto zámky musel rozmístit v programu sám. Pokud se píše aplikace zcela od začátku, tak by to takový problém nemusel být. U refaktorizace stávajících aplikací by se hodil nějaký automatizovaný způsob, ale to by bylo pravděpodobně velice náročné, protože by se musel přesně sledovat tok předávání ukazatelů.

7.1.2 Nesprávné uvolňování paměti

Mezi tyto problémy patří předčasné uvolnění paměti nebo dvojité uvolnění. První ze jmenovaných problémů se řeší poměrně obtížně a je nejčastěji zachycen jako nesprávný přístup k paměti, který jsem popsal v předchozí kapitole. Dvojité uvolnění paměti se řeší poměrně standardním způsobem, jelikož si alokátory vedou záznamy o volných a alokovaných úsecích paměti a pokud se aplikace pokouší uvolnit již uvolněný blok, je okamžitě hlášena chyba. Může se také stát, že se při nesprávném uvolňování paměti aplikace náhodou strefí do správně alokovaného úseku paměti. Nicméně jakmile se právoplatný vlastník pokusí tuto paměť uvolnit, dojde opět k případu dvojitého uvolnění paměti. Jelikož jsou tyto problémy řešeny standardně, nedefinuji žádné makro, které by tuto kontrolu povolovalo.

7.1.2.1 Alokátor velkých objektů

Tento alokátor nejdříve získá číslo stránky, na které daný blok začíná. Poté se snaží načíst informace asociované s touto stránkou. Pokud s touto stránkou nejsou asociované žádné informace, tak se nejspíše aplikace snaží uvolnit adresu, která nebyla vrácena funkci malloc. Pokud je deskriptor bloku získán, tak se musí zkontrolovat, jestli byl opravdu alokován. Jelikož je deskriptor asociován s první a poslední stránkou, tak se musí ještě zkontrolovat, jestli adresa pochází z první stránky. Následující úsek kódu tento text vysvětluje:

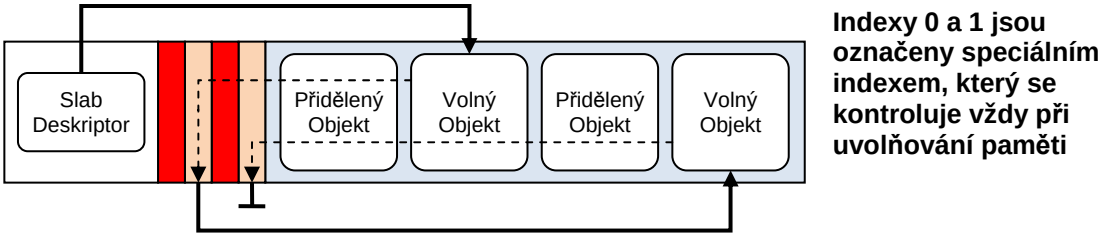
```
UInt32      dwPage = (pFreeMem - pMemRegion->pBase)/PAGE_SIZE;
PMEMBLOCK   pMemBlock = pMemRegion->blockAllocator[dwPage];
if (!pMemBlock || pMemBlock->bFree || pMemBlock->iPage != dwPage) {
#ifdef DEBUGMODE
      DbgOutput("Memory pointer points to free memory block");
#endif
      __debugbreak();
      goto _err_exit;
}
```

7.1.2.2 Alokátor malých objektů

U alokátoru malých objektů je situace o něco málo složitější. Problém především vzniká, pokud se používají vyrovnávací paměti, kde není prováděna žádná kontrola a vrácená adresa je okamžitě přidána do cache. K detekci dojde až je cache uvolňována, ale to už je pozdě, protože pokud se v cache nachází 2 stejné adresy, tak při alokaci může dojít k přidělení stejného objektu dvěma nezávislým voláním. Tento problém lze řešit dvěma způsoby. Buď se velikost všech cache sníží na hodnotu 1 a tím ji v podstatě vyřadíme z provozu nebo se vždy při vkládání adresy do cache zkontroluje, jestli se tam již nenachází. To může celkový výkon zpomalit, ale pokud je tato dodatečná kontrola nezbytná, lze aktivovat pomocí makra `MEMDOUBLEFREE`.

Kontrola dvojitého uvolnění paměti ve Slab cache je již poměrně přímočará. Slab si udržuje indexy volných bloků, které vytváří lineární seznam. Alokované objekty jsou z tohoto seznamu vyňaty a lze jim přiřadit jedinečný index, který je rozliší. Pokud se provádí uvolňování, musí index

tohoto objektu odpovídat této speciální hodnotě (BUFCTL_FREE). Následující obrázek 7.3 lépe ilustruje tento problém:



Obrázek 7.3: Detekce dvojitého uvolnění paměti ve Slab cache

Nastavování a kontrola této speciální hodnoty v odladěné aplikaci nemá žádný význam a proto se musí povolit již zmíněným makrem MEMDOUBLEFREE.

7.1.3 Únik paměti (memory leak)

Dalším velice rozšířeným problémem je špatné uvolňování paměti, které způsobuje vyčerpání dostupné paměti. Tohoto problému si často programátor okamžitě nevšimne, protože program funguje správně a nezpůsobuje žádné výjimky. Jakmile ale aplikace běží delší dobu, tak se může špatné uvolňování projevit snižováním volné paměti nebo nadměrným swapováním, což vede k degradaci výkonu celého počítače. Tuto kapitolu rozdělím do dvou částí: prostá detekce úniku paměti na úrovni alokátorů (makro LEAKSCHECK) a zjištění vlastníka neuvolněné paměti (MEMORYCHECK).

7.1.3.1 Alokátor velkých objektů

Tento alokátor udržuje seznam přidělených bloků, a pokud není tento seznam na konci aplikace prázdný, je nahlášen únik paměti. Alokátor zároveň zkontroluje, jestli jsou třídy volných bloků ve stavu, který se nacházel po inicializaci a zjistí se, jestli jsou všechny přidělené regiony prázdné. Takto zajistíme i kontrolu správné funkčnosti alokátoru.

7.1.3.2 Alokátor malých objektů

Alokátor je tvořen z části Buddy systémem a nad ním je Slab alokátor. Kontrolu na únik paměti provádím u obou především z důvodu ověření jejich sprívné funkčnosti. Uživatelské paměti se týká ale pouze část Slab alokátoru. U něj je kontrola prováděna tak, že se v cyklu projdou všechny vytvořené SizeCache, které se vždy skládají ze tří seznamů Slab cache (slabsPartial, slabsFull, slabsFree). Stačila by kontrola pouze na neprázdnost seznamů slabsPartial a slabsFull, ale jelikož chceme znát i adresu neuvolněné paměti, musíme ji v těchto Slab cache nalézt. To se provede tak, že se v každé Slab cache zkontrolují všechny indexy, jestli neobsahují speciální hodnotu indikující přidělený objekt (viz obrázek 7.3). Buddy systém zkontroluje, jestli jsou všechny seznamy volných bloků ve stavu po inicializaci. Je to velice podobné jako u alokátoru velkých objektů.

7.1.3.3 Pokročilá kontrola paměti

Předchozí metody zajišťovaly pouze detekci úniku paměti, ale pokud jsme chtěli zjistit, komu tento úsek patří, tak to nebylo možné. Nejdříve jsem vytvořil modul *memcheck* pro kontrolu samotných alokátorů, abych mohl zjistit, že se nikdy nevrátí dvě stejné adresy nebo dokonce překrývající se oblasti paměti. Poté jsem tento modul zobecnil a použil ho k trasování uživatelem alokované paměti. Nyní obsahuje záznamy: kdo tuto paměť alokoval, kdy, které vlákno, jak je tato paměť velká apod. Díky tomu, že je zde veden i záznam o tom, kterému vláknu tato paměť patří, tak lze kontrolovat únik paměti při ukončování jednotlivých vláken. Je ale zapotřebí, aby si jednotlivá vlákna nepředávala alokovanou paměť. Pokud je tato podmínka splněna, můžeme průběžně kontrolovat únik paměti a nečekat na ukončení celé aplikace.

Tento modul poskytuje rozhraní k jeho inicializaci, dále k registraci záznamů o alokované paměti a nejdůležitější část je funkce pro získání záznamu o úseku paměti ze zadané adresy. Funkce pro registraci úseku jsou následující:

```
AddUsedRegion(void* pAddress, int dwSize, MEM_ALLOCATOR dwType,
              char* pszFile, int dwLine, char* pszFunction);
DelUsedRegion(void* pAddress, int dwSize, MEM_ALLOCATOR dwType);
```

Pomocí nich se přidělená adresa paměti zaregistruje při alokaci a při uvolňování se opět odstraní. Při registraci je nutné vyplnit také soubor, číslo řádku a jméno funkce, ze které byla alokace volána. K tomu je možné použít preprocesor jazyka C, aby nemusel uživatel tyto údaje vyplňovat sám. Navíc by to bylo velice nepohodlné, když existují vždy dvě verze funkcí pro alokaci. Jedna pro verzi sloužící k ladění a druhá pro cílový překlad. Následující příklad ukazuje definici funkce malloc:

```
#ifndef MEMORYCHECK
#     define malloc    malloc_dp
      extern "C" void* malloc(size_t size);
#else
#     define malloc(size) \
             malloc_mc(__FILE__,__LINE__,__FUNCTION__,__VA_ARGS__)
      extern "C" void* malloc_mc(char* pszFile, int dwLine,
             char* pszFunction, size_t size);
#endif
```

Přidané parametry jsou umístěny před původními kvůli problému, který jsem řešil, když funkce obsahuje proměnný počet parametrů, způsobený například defaultními hodnotami parametrů. Tímto způsobem budou nahrazeny všechny funkce správně, protože standardní makro `__VA_ARGS__` dokáže zkopírovat všechny původní parametry a není omezeno nějakým konstantním počtem.

Následující funkce slouží k získání záznamu o úseku ležící na dané adrese:

```
PDBG_USEDREGION GetUsedRegion(void* pAddress);
```

Vlastní implementace jsem se snažil provést co nejefektivněji, aby se tato kontrola dala používat při normálním běhu aplikace a přitom ji příliš nezpomalovala. Paměť jsem se nesnažil za každou cenu ušetřit, protože by to mohlo vést ke zpomalení. Ale jelikož je tato kontrola prováděna pouze u aplikací sloužících k ladění, tak se tento problém příliš neprojeví. Pro alokaci vnitřních struktur jsem použil optimální alokátor, který jsem popsal v kapitole 5. Přidělenou paměť nerozšiřuji, a proto musí být na začátku nastavena na dostatečnou velikost. K synchronizaci jsem opět použil spinlock, který je popsán v kapitole 4.

Aby se k jednotlivým záznamům dalo rychle přistupovat, rozdělil jsem celý adresový prostor na úseky o velikosti jedné stránky. Jelikož se v tomto úseku může vyskytovat více objektů, je každý úsek tvořen lineárním seznamem, který se již prohledává sekvenčně. Záznam popisující celou paměť se tedy skládá z (2^{32} / 4096) seznamů, které jsou na počátku prázdné a postupně se plní při registraci úseků paměti. Ve standardním režimu se záznam vytvoří pouze na první stránku, která daný úsek začíná, což je mnohem rychlejší než zaznamenávat všechny použité stránky. Při plné kontrole se zaznamenávají všechny stránky a zároveň se ověřuje překrývání paměti.

Mezi další rozšíření bych chtěl přidat ke každému záznamu také trasování zásobníku. Díky tomu by programátor znal přesné volání funkcí v době přidělení paměti. Pokud je aplikace přeložena k ladění, oba operační systémy poskytují funkce, které umožní zjistit volané funkce ze zásobníku. Systém Windows používá *StackWalk64()* a sytém Linux *backtrace()*.

7.1.4 Neošetření návratové hodnoty NULL

Následující problém se týká nejvíce aplikací, které pracují s velkými objemy dat a zpracovávají obecně libovolně velká vstupní data. Každopádně je dobrým zvykem kontrolovat návratovou hodnotu funkce *malloc* v libovolné aplikaci. Pokud se tak nečiní a tato funkce vrátí NULL, obvykle aplikace zhavaruje na špatný přístup do paměti. I v případech kdy se tato hodnota kontroluje, může docházet k problémům. Typickým příkladem jsou inicializační rutiny, které postupně provádí několik alokací paměti a při výskytu chyby se tato inicializace musí přerušit a všechnu doposud přidělenou paměť zpět uvolnit. Pokud je tato inicializace složitá, tak se často stává, že se některá paměť zapomene uvolnit nebo se na tuto chybu reaguje špatně, což může opět způsobit výjimku.

V následujícím textu popíši dvě metody, které jsem implementoval k simulaci nedostatku paměti. Díky nim lze odhalit některé chyby, které způsobují pády aplikace. Obě metody se aktivují makrem `NULLGENERATOR`.

7.1.4.1 Náhodné generování NULL

U této metody je zcela na začátku funkce *malloc* umístěn následující kód:

```
if (GenerateNull()) return NULL;
```

Funkce *GenerateNull()* je pouze generátor rovnoměrného rozložení, který s určitou pravděpodobností generuje TRUE nebo FALSE. Největší výhodou této metody je, že se dá použít za běhu aplikace na libovolných vstupech bez dalších zásahů do algoritmu.

7.1.4.2 Systematické generování NULL

Předchozí metoda je poměrně jednoduchá, ale často nepostihuje celý stavový prostor všech možností, do kterých se může algoritmus dostat. Proto jsem se snažil vytvořit nějakou metodu, která se pokusí o systematický přístup. K tomuto účelu jsem vytvořil algoritmus založený na vyhledávacím algoritmu Breadth-first Search (BFS).

Pravděpodobně by zde bylo vhodné vložit celý zdrojový kód tohoto algoritmu, ale z úsporných důvodů nechám na čtenáři, aby ho vyhledal na přiloženém CD. Pokusím se tedy pouze o jeho slovní popis. Předem musím říct, že aby tento algoritmus fungoval, musí programátor přijmout určitá omezení. Zaprvé je nutné přichystat vstupní data, která v aplikaci způsobí projití co nejvíce větví programu, kde se používá alokace paměti. A zadruhé je nutné tato data opakovaně volat na stejný úsek kódu a zajistit, aby byl prováděn sekvenčně. Toto omezení se může zdát poměrně velké, ale v rámci testování jednotlivých modulů (*unit testing*) je to dosažitelné. Následující pseudokód ukazuje použití tohoto generátoru:

```
InitNullGenerator();                // inicializace generátoru
const inputData;
do {
      UnitTest(inputData) {
      ...
            malloc() {
                  if (GenerateNull()) return NULL;
            ...
            }
      ...
      }
} while (NextNullIteration());      // postupné zkoušení všech možností
```

Nyní si stručně představíme, jak tento algoritmus funguje. V prvním průchodu při každém volání *GenerateNull()* se pouze vytváří seznam všech volání *malloc*. Jelikož je algoritmus sekvenční, pořadí volání této funkce jednoznačně určuje pozici v algoritmu. Není tedy zapotřebí znát např. jméno funkce apod. Nicméně se tyto informace také udržují, aby programátor viděl, která kombinace volání funkcí způsobila problém. Poté co skončí tato inicializační fáze, je poprvé volána funkce *NextNullIteration()*, která z takto vytvořeného seznamu vybere první prvek a nastaví u něj, aby při dalším volání funkce *GenerateNull()* způsobil vygenerování NULL. Protože návrat hodnoty NULL nezpůsobí ukončení dalšího volání funkce *malloc*, musí se začít vytvářet seznam druhé úrovně všech volání funkcí *GenerateNull()*. Tato druhá úroveň se vytváří tak, že se jako předek nastaví funkce, která vyvolala NULL a poté se takto získaný prvek uloží na konec celého seznamu, abychom simulovali algoritmus BFS. Jakmile se projde celá první úroveň stromu, což odpovídá vyzkoušení všech možností jednoho generování NULL, je postupně procházena druhá úroveň, která odpovídá

všem možnostem generování dvou výpadků paměti. Takto je postupováno do té doby, než je celý zásobník vyprázdněn a jsou vyzkoušeny všechny možnosti.

Teoreticky tento algoritmus funguje dobře, ale je poměrně náročný na paměť, jelikož stavový prostor může být relativně velký. Místo algoritmu BFS lze použít i algoritmus *Depth-first search* (DFS), který poskytuje stejné výsledky, ale v některých případech může snížit spotřebu paměti. Je to z důvodu snižování počtu volání malloc, při více výpadcích paměti. To vyplývá z faktu, že se algoritmus nedokáže vypořádat s tolika chybami a snaží se výpočet co nejrychleji ukončit. Metodu BFS jsem zvolil kvůli tomu, že nejvíce chyb vzniká již na prvních úrovních a díky tomu je nalezneme mnohem rychleji.

7.2 Statistiky alokátoru

Při měření výkonu aplikace často chceme znát i využití systémové paměti, fragmentaci paměti, rychlost přidělování a uvolňování paměti apod. Proto implementace umožňuje volitelně zapnout pomocí makra `MEMORYSTATS` záznam těchto informací. Pomocí funkce *malloc_stats()* je možné za běhu veškeré informace zobrazovat.

Protože je často potřeba optimalizovat spotřebu paměti, je dobré znát funkce využití paměti v závislosti na jednotlivých funkcích. Proto jsem nad modulem memcheck vytvořil profiler, který se snaží shromažďovat co nejvíce agregovaných údajů o používání dynamické paměti u jednotlivých funkcí. Takže se shromažďuje: kolik paměti daná funkce celkově spotřebovala, kolikrát volala alokaci, jak dlouho tuto paměť používala atd. Protože funkce může volat více různých alokací, je vhodné, aby se co nejvíce rozlišily a zároveň se neukládalo příliš mnoho dat. Proto jsem vytvořil několik tříd velikostí alokací, do kterých jsou agregovaná data ukládána. Třídy jsou odvozeny z násobků dvou a díky tomu je dávána větší priorita malým objektům, které jsou tak rozlišeny mnohem přesněji. Nicméně toto chování bylo vytvořeno záměrně, protože je obecně alokováno více malých objektů. Agregovaná data se díky tomu ukládají do 32 tříd velikostí. Určení této třídy ze zadané velikosti vyžaduje spočítat dekadický logaritmus o základu 2. Aby tato operace nebyla příliš náročná, použil jsem instrukci BSR, která je dostupná na procesorech x86 a která dokáže rychle vyhledat pozici bitu nastaveného na 1. Jinými slovy to odpovídá nalezení dekadického log2. Na odlišných procesorech se musí použít jiná instrukce nebo klasický výpočet s cyklem.

Seznam všech funkcí, které volaly alokaci paměti, je udržován v asociativním poli typu hash table. Díky tomu, že se k získání jmen funkcí používá makro generátor, je téměř u všech překladačů zajištěno, že ukazatel na řetězec s názvem funkce bude vždy jediný. Proto jsem vytvořil hash funkci, která počítá index do tabulky z adresy ukazatele a ne ze skutečného řetězce. Tak je vyhledání tohoto záznamu o něco rychlejší. Nicméně problém dělají inline funkce, která často mají jiný ukazatel na stejný název. Proto je možné pomocí makra `PROFILER_STRINGMODE` povolit klasické počítání indexu hash tabulky.

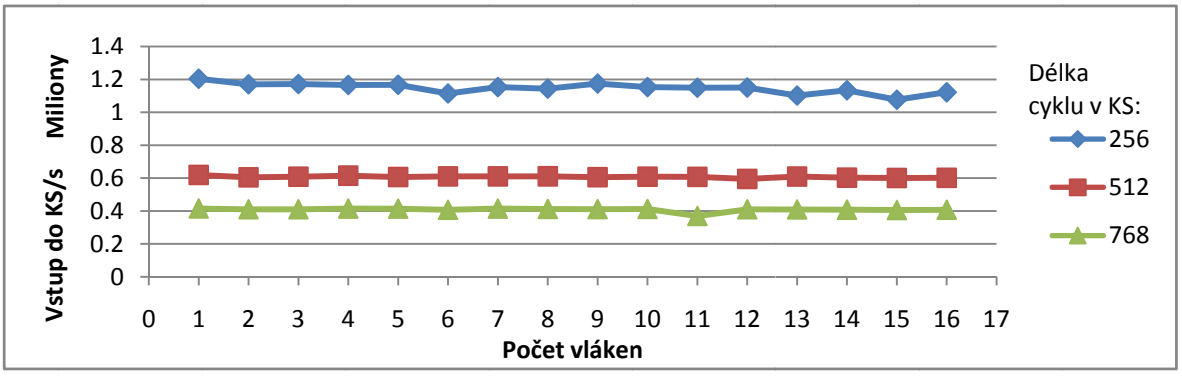
8 Srovnávací testy

V této příloze popíši výsledky testů, které jsem provedl na různých systémech za účelem porovnat stávající správce paměti se správcem, který jsem vytvořil v rámci této diplomové práce. Jelikož jsem doposud tento nový alokátor nepojmenoval, budu ho v dalším textu označovat jako `malloc_dip`.

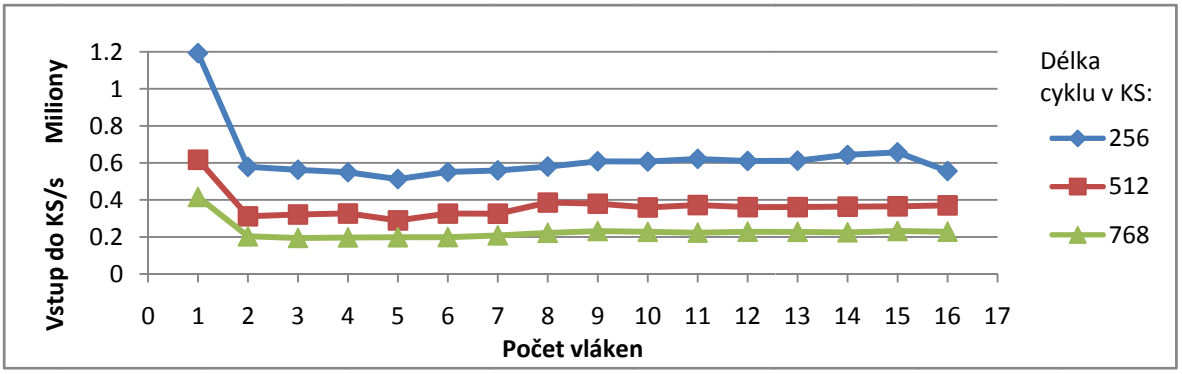
8.1 Synchronizace

Nejdříve jsem provedl měření zámku Spin lock, který jsem popisoval v samostatné kapitole, se standardními zámky dodávané s operačním systémem. Testy jsem prováděl především na pracovním serveru, ke kterému jsem měl přístup, ale bohužel byl na něm nainstalovaný pouze operační systém Microsoft Windows Server 2003 Enterprise x64 Edition. Nicméně to je velice moderní a výkonný stroj. Jeho parametry jsou následující: 2x Intel Quad-Core Xeon E5335 2.0 GHz CPU (8 procesorů), 8 GiB RAM.

Testy byly prováděny tak, že se použil jeden globální zámek, pomocí něhož se jednotlivá vlákna pokoušela soupeřit o vstup do kritické sekce. Jakmile se do této sekce některé dostalo, následoval předem daný počet cyklů, který prováděl pouze inkrementaci a měl simulovat výpočet. Počet vstupů do kritické sekce byl také předem daný a po jeho dosažení se měřil uplynulý čas.



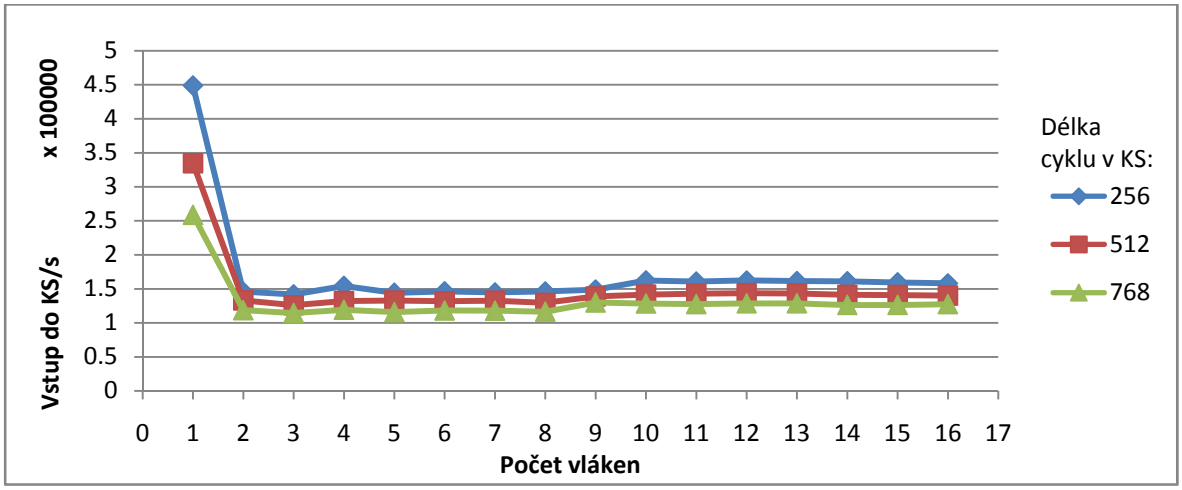
Graf 8.1: Výsledky měření zámku *Spin lock* na systému Windows



Graf 8.2: Výsledky měření zámku *Critical Section* na systému Windows

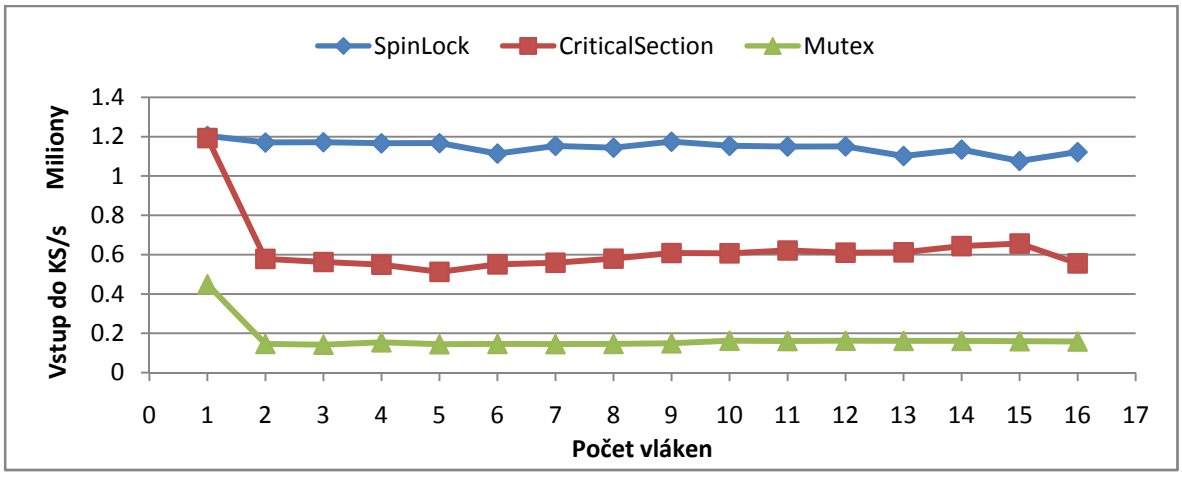
K synchronizaci vláken na systému Windows je nejvhodnější použít zámek CritialSection. Proto jsem především nový Spin lock testoval s tímto zámkem. U standardního alokátoru je vidět dvojnásobné snížení výkonu, jakmile je použito více jak jedno vlákno a dochází k soupeření o zámek. Spin lock se choval mnohem stabilněji a s přibývajícím počtem vláken zůstával výkon relativně konstantní. Navíc umožnil dvojnásobný počet vstupů do kritické sekce. Dále je možné vidět, že prodlužování délky cyklu se chová standardně a tento zámek nezpůsobuje problémy i při delším čekání. Proto jsem se v mé implementaci správce paměti rozhodl tento zámek použít.

Spíše pro zajímavost jsem testoval i Mutex (viz graf 8.3), používaný v systému Windows. Z grafu je vidět že jeho výkon je 12x horší a tento zámek se dá použít pouze k dlouhodobému vyloučení nebo pokud je nutné sdílet zámky mezi jednotlivými procesy. Naproti tomu u systému Linux se mutex chová podobně jako zámek Critical Section u systému Windows a proto se na tomto systému používá nejčastěji. Toto chování je způsobeno spíše různými názvy téhož druhu zámku.



Graf 8.3: Výsledky měření zámku *Mutex* na systému Windows

Následující graf ukazuje porovnání jednotlivých zámků, kde byl použit cyklus v kritické sekci o délce 256:



Graf 8.4: Srovnání jednotlivých zámků na systému Windows (*délka cyklu v KS je 256*)

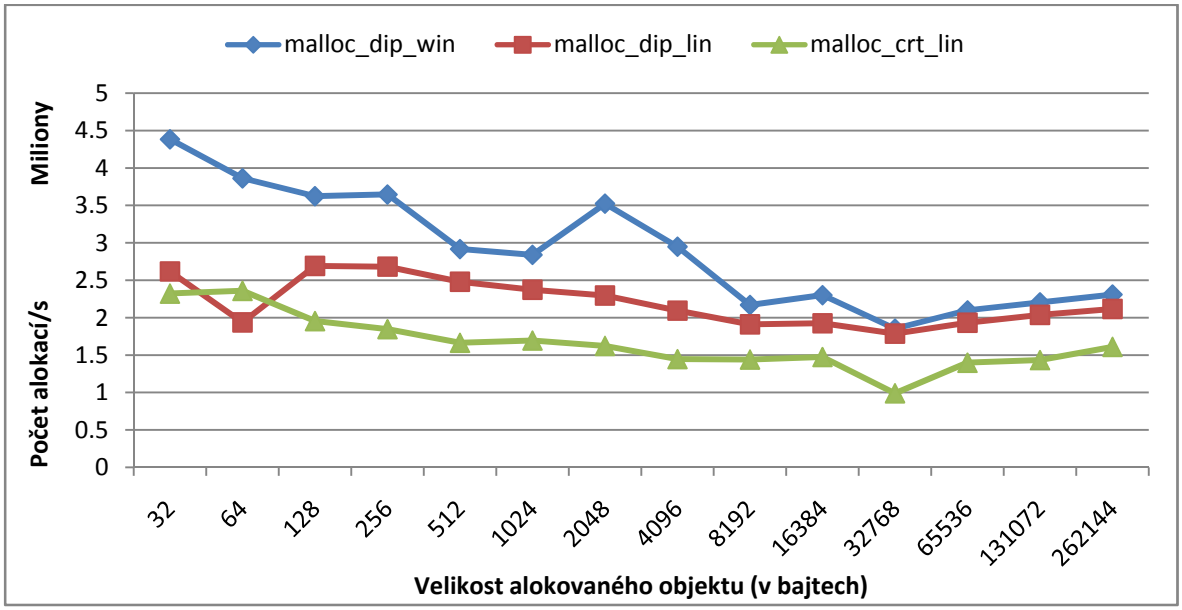
8.2 Porovnání správců paměti

Z důvodu objektivity jsem provedl testy mého správce paměti na jednom počítači v operačním systému Linux a Windows. Jednalo se o jednoprocesorový desktop s procesorem AMD Athlon 64 3000+ a 2 GiB RAM. Dále jsem provedl samostatné testy na serverové stanici s Windows Server, kterou jsem popsal v kapitole o synchronizaci.

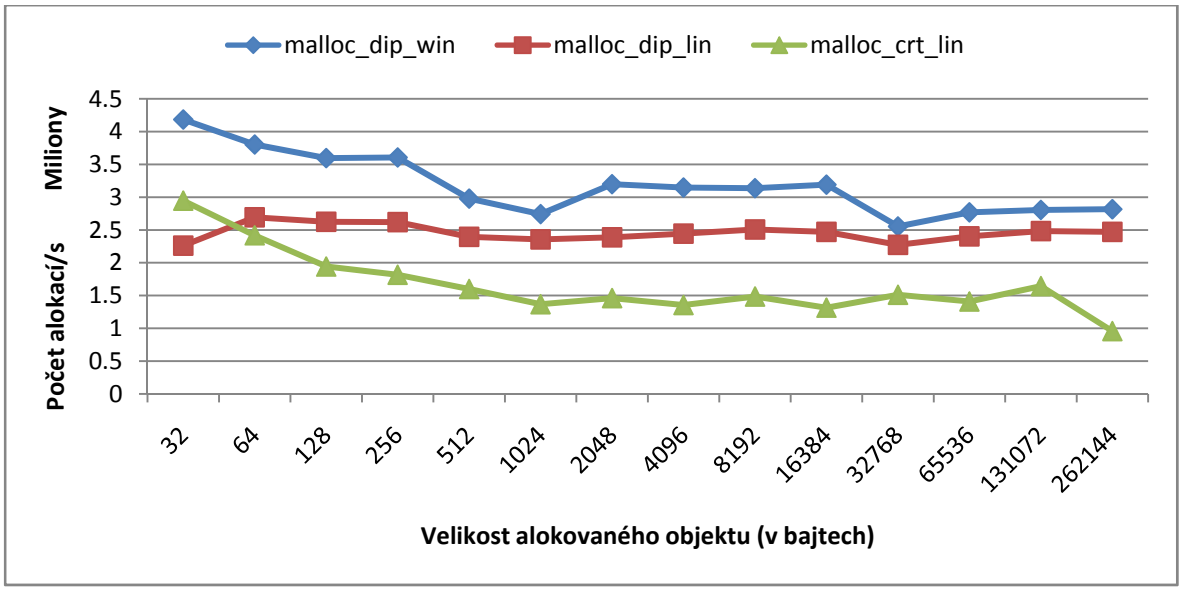
Testy byly prováděny obdobně jako testy synchronizace. Hlavním rozdílem bylo, že se neměřil počet vstupů do kritické sekce, ale počet alokací a uvolnění paměti za jednotku času. Dále se v těchto testech nesimuloval výpočet a tyto alokace se vůbec neomezovaly. Na začátku se každému vláknu vypočítal maximální počet udržovaných ukazatelů na paměť tak, aby celkové využití paměti bylo maximálně 64 MiB. Dále se rozdělil počet alokací tak, aby celkový počet činil 5 milionů alokací a uvolnění paměti. Toto číslo jsem musel u standardního alokátoru Windows snižovat, abych byl schopný dostat nějaké výsledky v únosném čase. Měření bylo prováděno vždy 4x pro různý počet vláken a různou velikost alokovaných objektů.

8.2.1 Standardní správce systému Linux

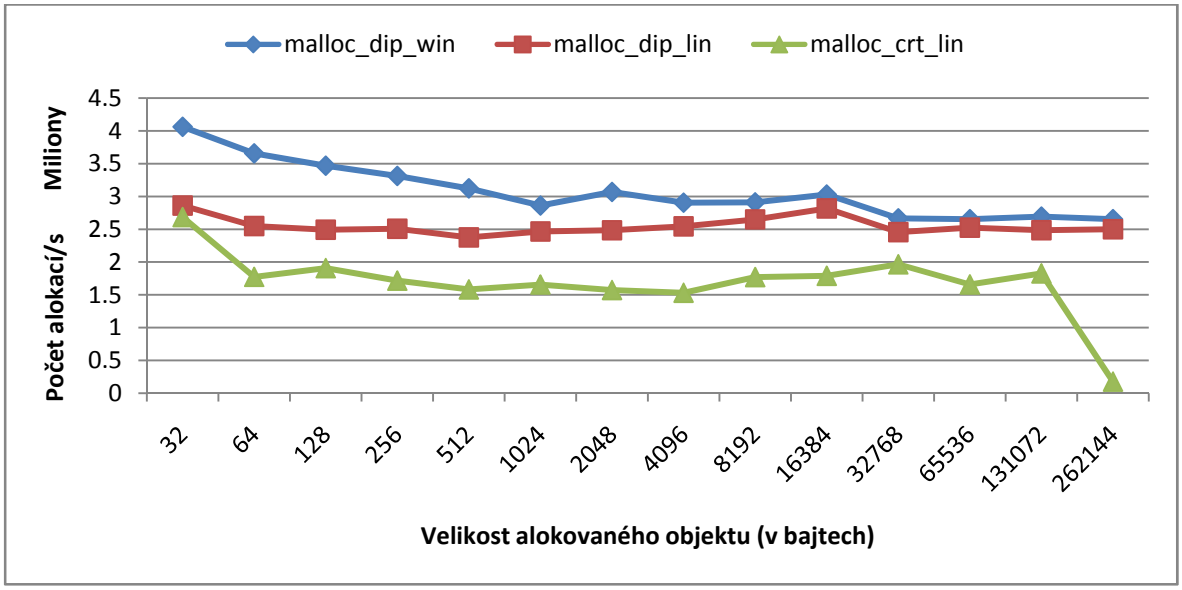
Po řadě testů jsem zjistil, že standardní správce paměti obsažený v CRT knihovně systému Linux je oproti systému Windows mnohem vyspělejší a bez větších problémů lze používat i v náročných aplikacích. Testy byly prováděny na jednom počítači. Ale jak je z grafů patrné, implementace `malloc_dip` na systému Windows byla o něco rychlejší. To může být způsobeno mnoha faktory. Například jsem použil kvalitní kompilátor firmy Intel, který se vyznačuje efektivní optimalizací. Také je možné, že systém Windows o něco lépe pracuje s vlákny. Na systému Linux byla podpora vláken dodávána až dodatečně a je převážně odvozena z procesů.



Graf 8.5: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*1 vlákno*)



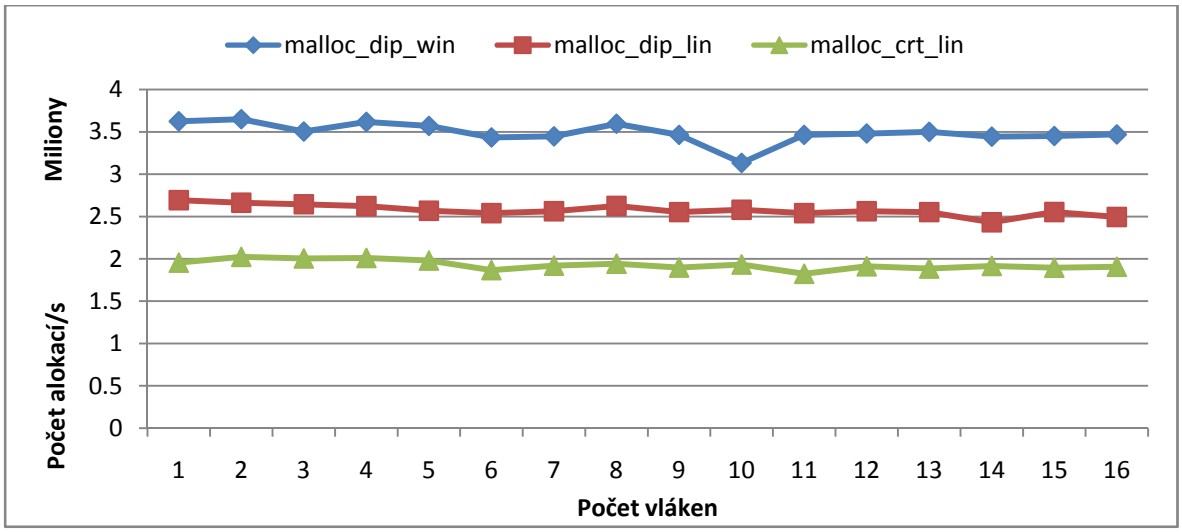
Graf 8.6: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*8 vláken*)



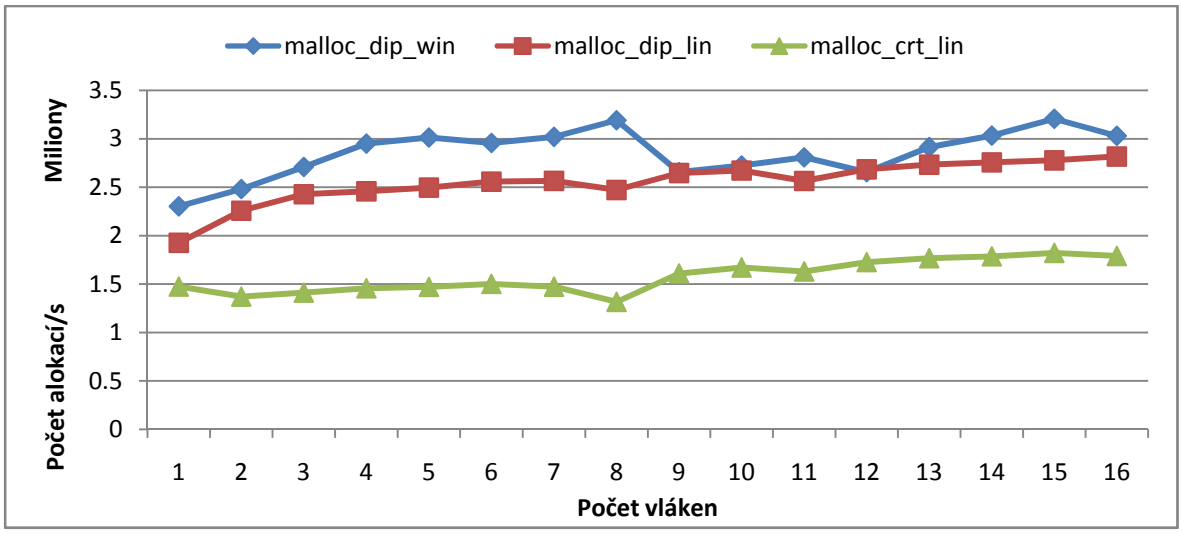
Graf 8.7: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*16 vláken*)

Z grafu je patrné, že malloc_dip dosahuje nejlepších výsledů pokud používá Slab alokátor. Nicméně pokles výkonu není u alokátoru velkých objektů nikterak výrazný. Implementace standardního alokátoru v CRT knihovně pracuje také v několika režimech. U menších objektů používá tzv. buckets, což je obdoba Slabu a k alokaci velkých objektů Best fit.

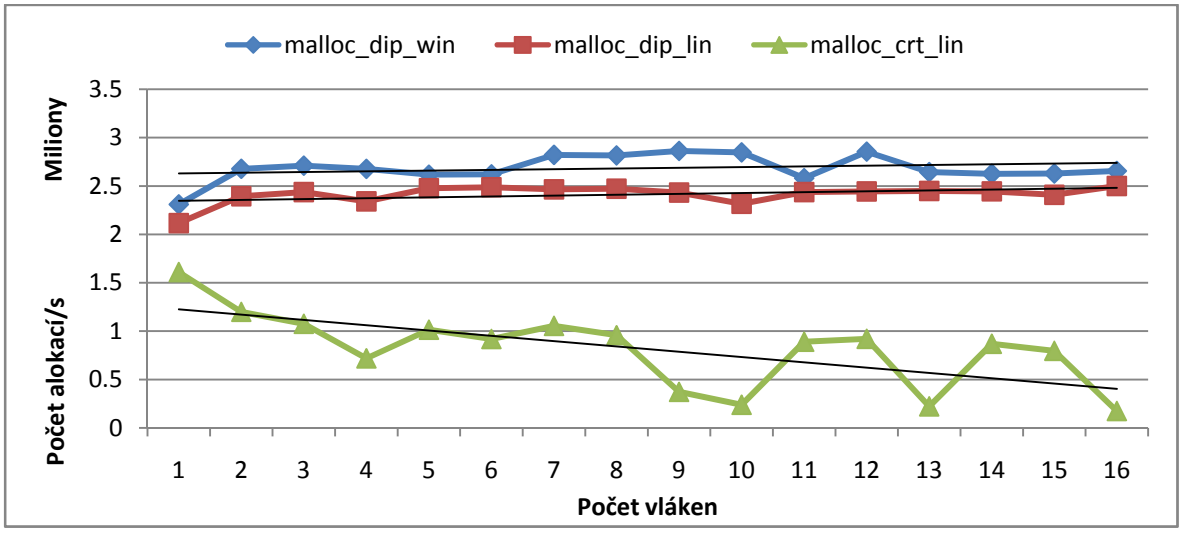
Následující grafy ukazují závislost počtu alokací objektů určité velikosti na počtu použitých vláken. Ideální průběh by byla neklesající křivka. U malých objektů tento průběh splňovaly všechny testy. Nicméně u standardního alokátoru v Linuxu se u větších objektů projeví algoritmus Best fit, který je o něco pomalejší a tato křivka klesá (viz graf 8.10). Každopádně má menší interní fragmentaci a je nutné zvážit, co od dané aplikace vyžadujeme.



Graf 8.8: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*objekt 128 B*)

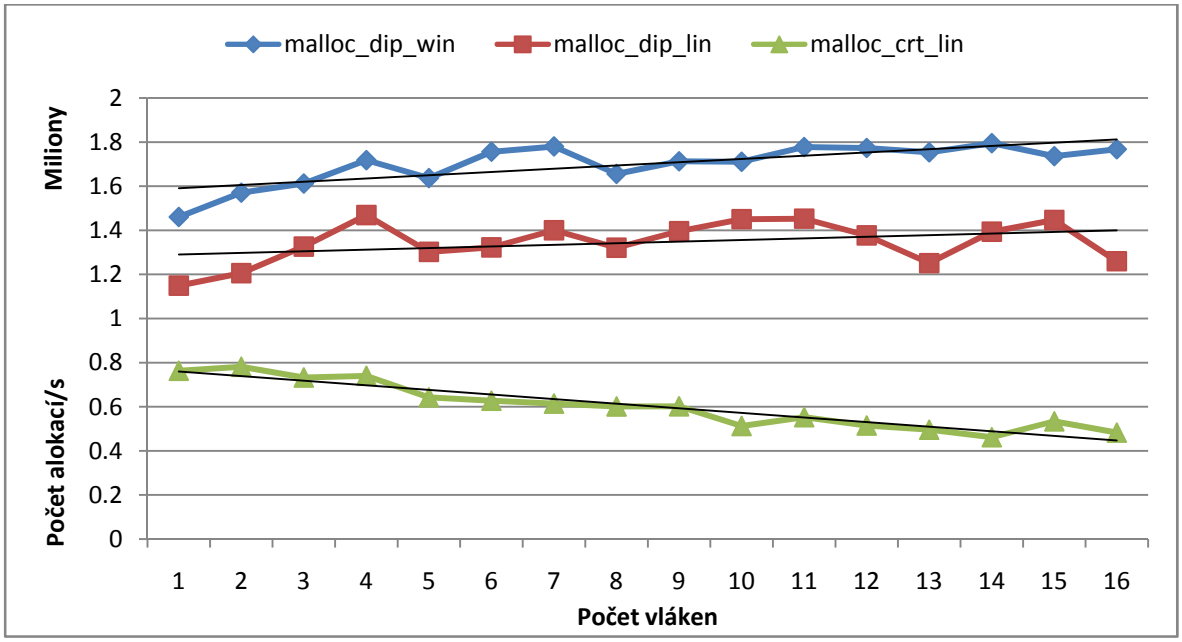


Graf 8.9: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*objekt 16 KiB*)



Graf 8.10: Srovnání malloc_dip na Windows a Linux s malloc na Linuxu (*objekt 256 KiB*)

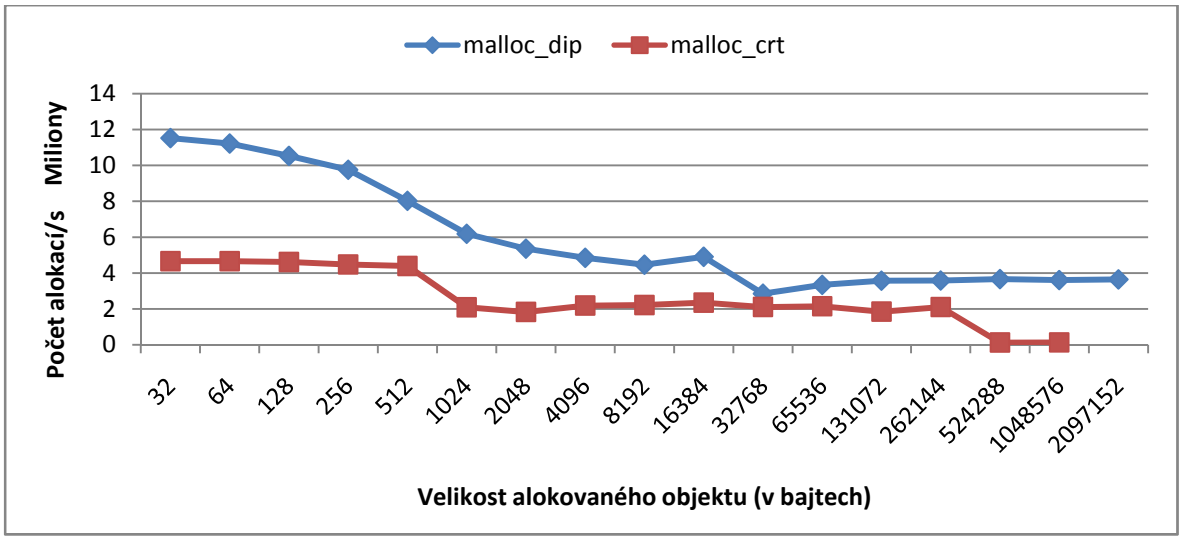
Na závěr jsem provedl nejspíše nejvýznamnější test. V něm jsem místo pevné velikosti alokovaného objektu použil náhodný generátor, který vytvářel požadavky na přidělení paměti o velikosti 31 bajtů až 134 KiB. Díky tomu se v alokátorech nevytvářely volné bloky o stejných velikostech, které uměle zvyšují rychlost alokace. Graf 8.11 opět ukazuje o něco horší výsledky standardního alokátoru, ale navíc je vidět, že jeho výkon rychleji klesá s počtem použitých vláken. U alokátoru malloc_dip je dokonce tato křivka rostoucí, což způsobují vyrovnávací paměti, kterých s počtem vláken úměrně přibývá.



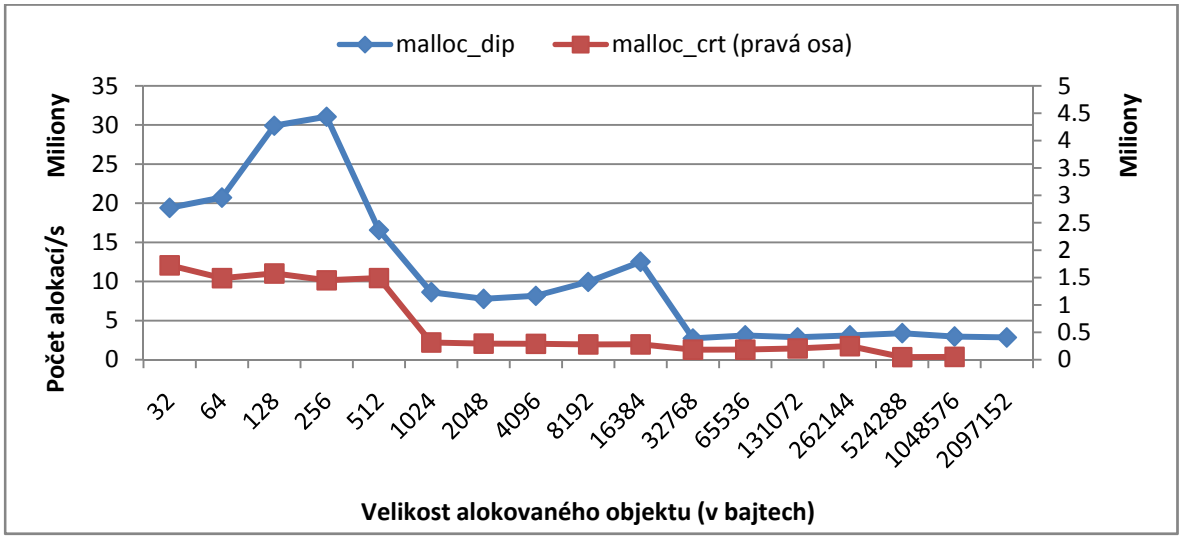
Graf 8.11: Srovnání malloc_dip s náhodně volenou velikostí alokovaného objektu

8.2.2 Standardní správce systému Windows

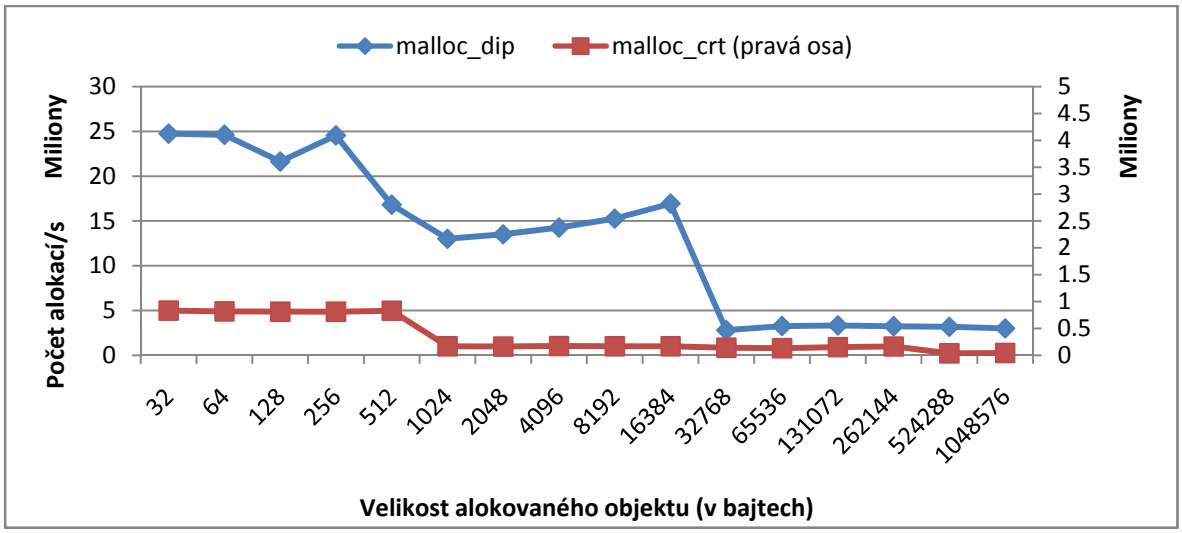
Nativní vývojové prostředí systému Windows je bezpochyby Microsoft Visual Studio. To obsahuje standardní CRT knihovnu pro jazyk C, které implementuje základní funkce. Mezi ně patří i správce paměti. Překvapením pro mě bylo, že CRT knihovna pouze slouží jako obal pro systémovou funkci HeapAlloc, která je implementovaná velice neefektivně, speciálně pro vícevláknové aplikace. Navíc existují dvě implementace. Jedna slouží pro standardní běh (*release*) a druhá pro ladění (*debug*). Bohužel jsou veškeré kontroly zdvojeny. CRT knihovna v Debug režimu provádí totiž podobné kontroly znovu a tím tak velice znatelně zpomaluje celý běh. Největším problémem je, že v tomto režimu se několikrát kontroluje celá oblast přidělované i uvolňované paměti a navíc se ještě vždy při úspěšném přidělení/uvolnění paměti celá oblast nastaví na určitou hodnotu. Děje se tak z důvodu ochrany paměti, ale způsobuje to velké zpomalení a náročnější aplikace se v tomto režimu špatně testují. Dalším problémem je, že se Release verze nesmí spouštět připojená k degugeru, protože ntdll takto detekuje Debug verzi a začne opět celou paměť nastavovat. Toto chování využívají některé programy jako jednoduchou ochranu proti ladění.



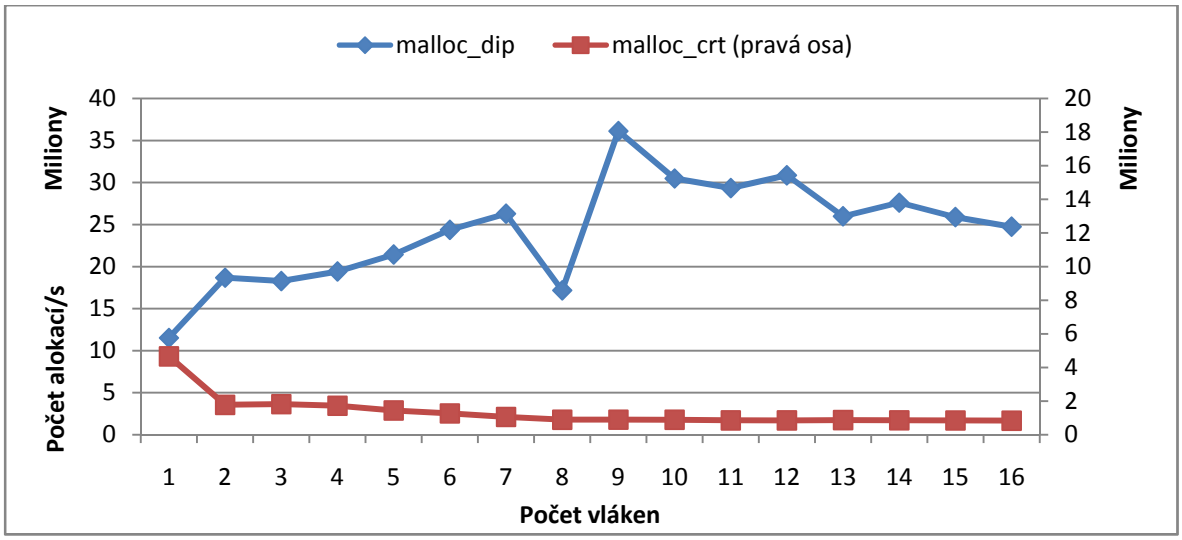
Graf 8.12: Srovnání malloc_dip s HeapAlloc na systému Windows (*1 vlákno*)



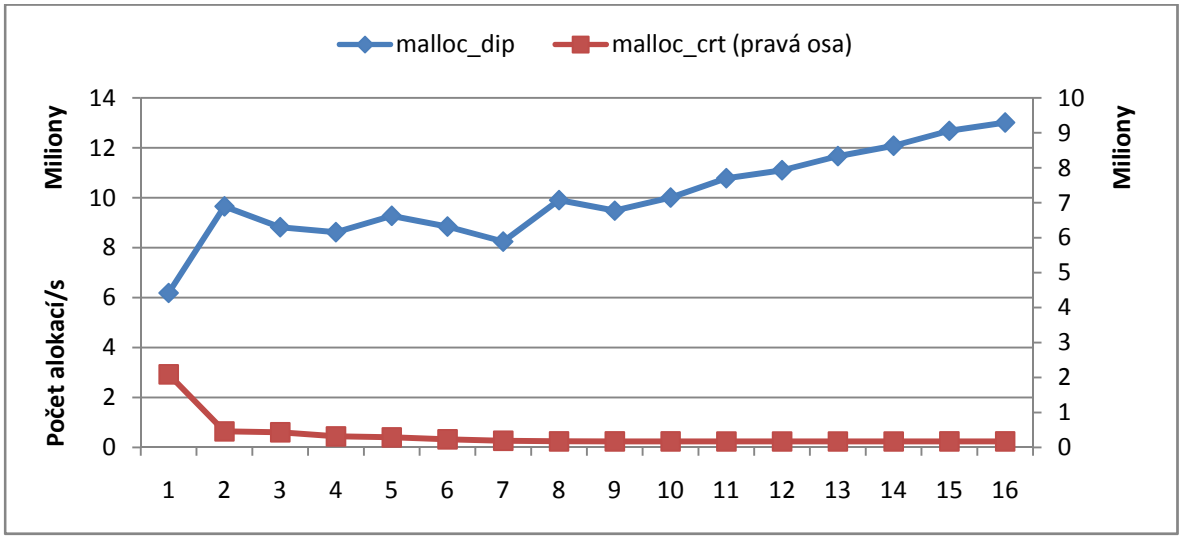
Graf 8.13: Srovnání malloc_dip s HeapAlloc na systému Windows (*4 vlákna*)



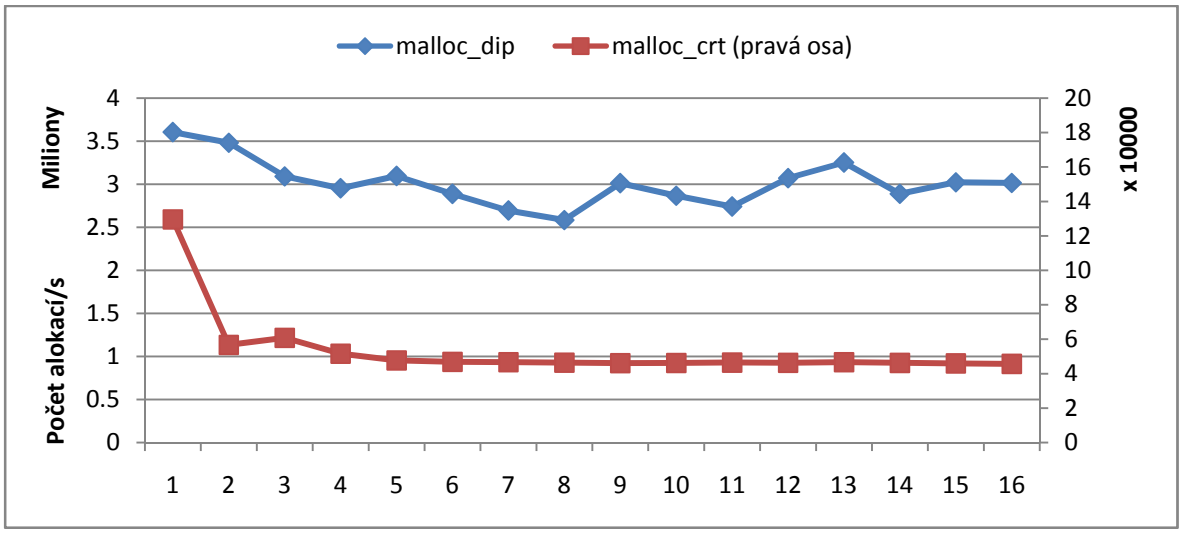
Graf 8.14: Srovnání malloc_dip s HeapAlloc na systému Windows (*16 vláken*)



Graf 8.15: Srovnání malloc_dip s HeapAlloc na systému Windows (*objekt 32 B*)



Graf 8.16: Srovnání malloc_dip s HeapAlloc na systému Windows (*objekt 1 KiB*)



Graf 8.17: Srovnání malloc_dip s HeapAlloc na systému Windows (*objekt 1 MiB*)

Předchozí grafy ukazují naměřené výsledky na systému Windows Server s osmi procesory. Abych byl schopný zobrazit výsledky HeapAlloc, musel jsem použít jiné měřítko grafu, které je zobrazeno na pravé svislé ose. Pokud je použito pouze jedno vlákno (graf 8.12), výsledky jsou relativně použitelné, ale jak ukazuje například graf 8.16, výkon velice výrazně poklesne s použitím více vláken. Graf 8.12 také prozrazuje, že standardní alokátor nejspíše používá 3 režimy. V grafu lze jasně rozpoznat úseky 0 až 512 B, 1 KiB až 256 KiB a 0,5 MiB a více, které vždy způsobí pokles výkonu. Toto chování lze pozorovat i za použití více vláken. Graf 8.17 ukazuje výrazný propad u velikosti objektu 1 MiB. Těchto objektů je možné alokovat už pouze 50 tisíc za sekundu. Jelikož tolik objektů naráz pravděpodobně nikdy alokovat nebudeme, nemusí se to zdát jako velký problém, ale svědčí to o nějakém vnitřním problému, který může způsobovat úzké hrdlo aplikace. Tento pokles přičítám přímému používání funkce VirtualAlloc, kterou tento alokátor na tak velké objekty přímo volá. Vedlejším efektem této funkce je totiž nulování celé alokované oblasti.

U alokátoru malloc_dip lze jasně pozorovat výhodu použití vyrovnávacích pamětí, které zajistí několikanásobný nárůst výkonu oproti přímému použití synchronizace. Výkyvy v rychlosti při použití různých velikostí jsem přesně neurčil, jelikož bych musel provést hlubší analýzu. Předpokládám, že to může být způsobeno různým vlivem úrovní cache. Nejvíce viditelná změna výkonu je od velikosti objektu 32 KiB. Od této velikosti se používá alokátor velkých objektů a je proto použita vždy synchronizace. Tento alokátor se od této velikosti chová poměrně konstantně a jeho výkon je stále dostatečný.

Závěr

V této práci jsem popsal různé principy správy virtuální paměti, bez jejichž hlubšího pochopení by nebylo možné přistoupit k vlastnímu návrhu efektivního alokátoru. Se znalostí určitých zákonitostí návrhu operačního systému a hardwaru, který s tím také velmi úzce souvisí, dokážeme lépe odhadnout možné problémy, které by vedly k celkovému snížení výkonu vlastního algoritmu alokace, popřípadě k nevhodně zarovnaným návratovým adresám alokovaných objektů. V další části práce jsem se zabýval konkrétními algoritmy manuálních správců paměti, které se dnes používají nebo jsou základem pro další složitější algoritmy.

Dále jsem popsal návrh a implementaci efektivního alokátoru, kterou jsem vytvořil v rámci této diplomové práce. Pří návrhu jsem vycházel z již existujících nebo mírně upravených algoritmů a pokusil se je spojit tak, aby společně vytvořily robustní správu paměti, která se dokáže přizpůsobit aktuálním požadavkům aplikace. Implementaci jsem se snažil vytvořit tak, aby byla co nejméně závislá na použitém operačním systému a přenositelnost byla zajištěna doplněním pouze nejnutnější množiny systémově závislých funkcí. Bohužel se mi několikrát stalo, že jsem musel zasahovat do již navržené architektury, jelikož byly některé systémové závislosti značně odlišné. Především nastal problém při zachytávání a zpracování ukončení vláken.

V práci jsem se také věnoval problémům, které vznikají při vývoji aplikace a úzce souvisí se správou dynamické paměti. Popsal jsem několik nástrojů, které některé z těchto problémů dokážou detekovat a případně usnadnit jejich odstranění. Většinu popsaných nástrojů se mi podařilo začlenit do implementace a jen některé zůstaly pouze ve stádiu návrhu. O jejich realizaci se budu snažit v dalším rozšíření projektu. Jelikož tyto nástroje zpomalují celkový výkon správce paměti, poskytl jsem možnost jak aktivovat pouze některé z nich nebo je zcela vypnout.

Tato práce je v příloze doprovázena také sérií testů, které jsem provedl v rámci srovnávacích studií s běžně používanými standardními alokátory na systémech Windows a Linux. Přesvědčil jsem se o tom, že systém Linux prošel mnohem delším vývojem v oblasti správy paměti než operační systém Windows. Proto bylo mým hlavním cílem vytvořit efektivního správce především pro serverové stanice se systémem Windows, na kterých jsme pozorovali pokles výkonu a měli jsme velké problémy s verzí programu určenou k ladění. V tomto ohledu jsem dosáhl značného zlepšení. Na systému Linux došlo také ke zlepšení, ale již nebylo tak výrazné. Díky tomu je v běžných aplikacích lepší používat standardní alokátor, který produkuje menší interní fragmentaci. Právě na zlepšení interní fragmentace bych se rád zaměřil v dalším rozvoji projektu. Řešení bude pravděpodobně spočívat v použití více tříd v alokátoru velkých objektů, které budou obsahovat mnohem menší objekty než doposud. Toto bude možné použít, jelikož k hledání volného bloku používám princip bitového pole, který snižuje počet cyklů k nalezení volné třídy.

Literatura

[1] Wikipedia: *Virtuální paměť* [online]. Poslední revize 2007-11-02 [cit. 2007-11-08]. <http://cs.wikipedia.org/wiki/Virtuální_paměť>.

[2] Wikipedia: *Virtual memory* [online]. Poslední revize 2007-11-05 [cit. 2007-11-08]. <http://en.wikipedia.org/wiki/Virtual_memory>.

[3] Lórencz, R.: *Architektura počítačových systémů* [online]. Přednáška, Katedra počítačů, FEL ČVUT Praha, 2005 [cit. 2007-11-20]. <http://service.felk.cvut.cz/courses/36APS/lectures>.

[4] Wikipedia: *Stránkování* [online]. Poslední revize 2007-11-03 [cit. 2007-11-12]. <http://cs.wikipedia.org/wiki/Stránkování>.

[5] Staudek, J.: *Virtuální paměť* [online]. Přednáška, Fakulta informatiky MU, [cit. 2007-12-03]. <http://www.fi.muni.cz/usr/staudek/vyuka/opsysprincipy/05_virtualni_pamet.pdf>.

[6] Wikipedia: *Segmentace* [online]. Poslední revize 2007-11-20 [cit. 2007-12-16]. <http://cs.wikipedia.org/wiki/Segmentace>.

[7] *Intel IA-32 Architectures Software Developer's Manual*. System Programming Guide, 2006.

[8] Wikipedia: *Paging* [online]. Poslední revize 2007-12-10 [cit. 2007-12-16]. <http://en.wikipedia.org/wiki/Paging>.

[9] Mrázek, L.: *Stránkování na žádost* [online]. Jihočeská Univerzita, [cit. 2007-12-19]. <http://homen.vsb.cz/~kod31/vyuka/opsys/strankovani.na.zadost.html>.

[10] Gill, B. S.: *Sequential Prefetching in Adaptive Replacement Cache*. IBM Almaden Research Center, 2005.

[11] Griffioen, J., Appleton, R.: *Reducing file system latency using a predictive approach*. USENIX Summer, 1994. s. 197-207.

[12] Curewitz, K. M., Krishnan, P., Vitter, J. S.: *Practical prefetching via data compression.* SIGMOD international conference, 1993. s. 257-266. ISBN 0-89791-592-5.

[13] Palmer, M. L., Zdonik, S. Fido: *A cache that learns to fetch*. 17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings, 1991. s. 255-264. ISBN 1-55860-150-3.

[14] Kroeger, T. M., Long, D. D. E.: *Design and implementation of a predictive file prefetching algorithm*. USENIX Annual Technical Conference, 2001. s. 105-118. ISBN 1-880446-09-X.

[15] Microsoft Developer Network: *Memory Protection* [online]. Poslední revize 2008-04-24 [cit. 2008-04-24]. <http://msdn2.microsoft.com/en-us/library/aa366785.aspx>.

[16] Russinovich, M., Solomon, D. A.: *Microsoft Windows Internals*. Fourth Edition, Microsoft Press, 2004. ISBN 0735619174.

[17] Wikipedia: *Memory hierarchy* [online]. Poslední revize 2008-03-13 [cit. 2008-03-18]. <http://en.wikipedia.org/wiki/Memory_hierarchy>.

[18] Patočka, M.: *Srovnávací studie jádra Linuxu a FreeBSD*. Diplomová práce, UK Praha, 2003.

[19] *Cache Organization* [online]. Poslední revize 2002 [cit. 2008-03-23]. <http://ece-www.colorado.edu/~ecen2120/Manual/caches/cache.html>.

[20] LWN.net: *CPU caches* [online]. Poslední revize 2007-10-01 [cit. 2008-03-24]. <http://lwn.net/Articles/252125/>.

[21] Hyde, R.: *The Art of Assembly Language*. No Starch Press, 2000. ISBN 1886411972.

[22] Beneš, M.: *Správa paměti* [online]. Katedra informatiky, VŠB-TU Ostrava, [cit. 2007-11-25]. <http://www.cs.vsb.cz/behalek/vyuka/pte/texty/pamet>.

[23] *The Memory Management Reference: Beginner's Guide* [online]. [cit. 2007-11-18]. <http://www.memorymanagement.org/articles>.

[24] Wikipedia: *Fragmentation* [online]. Poslední revize 2007-11-20 [cit. 2007-11-22]. <http://en.wikipedia.org/wiki/Fragmentation_(computer)>.

[25] Čermák, I.: *Přidělování paměti do nesouvislých úseků* [online]. Výpočetní centrum VŠE. [cit. 2007-12-16] <http://nb.vse.cz/~cermaki/os2.htm>.

[26] Blunden, B.: *Memory Management: Algorithms and Implementation in C/C++*. Wordware Publishing, 2003. ISBN 1556223471.

[27] Knuth, D. E.: *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing, 1998. ISBN 0-201-89685-0.

[28] Wikipedia: *Buddy memory allocation* [online]. Poslední revize 2007-12-11 [cit. 2007-12-20]. <http://en.wikipedia.org/wiki/Buddy_memory_allocation>.

[29] Wikipedia: *Slab allocation* [online]. Poslední revize 2007-11-28 [cit. 2007-12-06]. <http://en.wikipedia.org/wiki/Slab_allocator>.

[30] Bonwick, J.: *The slab allocator: an object-caching kernel memory allocator*. USENIX Summer Technical Conference Volume 1, 1994. 6 s.

[31] Jones, M. T.: *Anatomy of the Linux slab allocator* [online]. Poslední revize 2007-05-15 [cit. 2007-12-08]. <http://www-128.ibm.com/developerworks/linux/library/l-linux-slab-allocator>.

[32] Bovet, D. P., Cesati, M.: *Understanding the Linux Kernel, 3rd Edition*. Oreilly & Associates Inc., 2005. ISBN 0596005652.

[33] SourceForge.net: *TCMalloc: Thread-Caching Malloc* [online]. Poslední revize 2003 [cit. 2008-03-25]. <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>.

Seznam příloh

Příloha 1. CD se zdrojovými texty

Příloha 2. Správa paměti OS Windows

Příloha 3. Správa paměti OS Linux

Příloha A – Správa paměti OS Windows

Tato příloha nemá sloužit jako referenční příručka a pouze shrnuje základní funkce, které poskytuje operační systém pro práci s pamětí.

Funkce hromady	
HeapAlloc/*HeapReAlloc*	Alokuje/Realokuje blok paměti z hromady. Hromada je asociována s procesem a před jejím použitím se musí vytvořit pomocí funkce HeapCreate. Ta bohužel poskytuje minimální možnosti nastavení. Jelikož je halda udržována pomocí standardní struktury HANDLE, která lze pomocí funkce DuplicateHandle předat jinému procesu, může být halda sdílena mezi více procesy.
HeapFree	Uvolnění bloku paměti z haldy.
HeapSize	Získání velikosti alokovaného bloku z haldy.
Funkce pro správu virtuální paměti	
VirtualAlloc	Funkce umožňuje rezervovat určitý počet stránek MEM_RESERVE. Pokud k těmto stránkám potřebujeme přistupovat, musí se tato funkce volat podruhé s parametrem MEM_COMMIT, který způsobí přidělení virtuálním stránkám fyzické rámce. Aby nemusela být volána funkce 2x, lze tyto operace spojit. Pokud není nastaven flag MEM_RESET, je přidělená paměť automaticky nulována. Na rozdíl od správce heapu neobsahuje funkci pro realokaci virtuální paměti a musí to být řešeno pomocí rezervování většího úseku paměti. Funkce může být s parametrem MEM_RESERVE volána pouze 65536 krát. Tato informace není nikde dokumentována a jedná se buď o chybu, nebo nějaké omezení!
VirtualFree	Funkce umožňuje odmapovat určitou oblast přidělených stránek pomocí flagu MEM_DECOMMIT nebo zcela uvolnit rezervovaný virtuální prostor pomocí MEM_RELEASE.
VirtualProtect	Změna přístupových práv k určité oblasti stránek. Přístupová práva lze nastavit pomocí funkce VirtualAlloc a tato funkce slouží pro jejich úpravu nebo zjištění aktuálního nastavení.
VirtualQuery	Umožňuje zjistit informace o všech úsecích stránek ve virtuální paměti, které byly asociované s volajícím procesem.
Funkce pro mapování souborů do paměti	
CreateFile	Původně tato funkce sloužila pouze k vytvoření nebo otevření souboru. V systému NT byl tento model obdobně jako v UNIXu zobecněn na libovolná zařízení.
CreateFileMapping	Pouze vytvoří mapovací objekt z objektu získaného pomocí CreateFile, který určuje přístupová práva a úsek, se kterým bude možné pracovat.
MapViewOfFile	Vytvoří pohled na mapovací objekt. To znamená skutečné namapování určité oblasti do virtuálního prostoru paměti. Počet pohledů je omezen pouze rozsahem adres ve virtuálním prostoru a mohou se navzájem překrývat.
UnmapViewOfFile	Uvolnění pohledu z virtuálního prostoru paměti.

Příloha B – Správa paměti OS Linux

Systém Linux se na rozdíl od systému Windows snaží co nejvíce snížit volání jádra a co nejvíce funkcí delegovat na aplikační úroveň. Proto je správce paměti vytvořen ve standardní knihovně jazyka C a nesnaží se tuto zodpovědnost delegovat na systém. Navíc práci s virtuální pamětí a mapování souborů do paměti neodděluje a chápe ji pouze jako jiné pohledy na tentýž věc. Proto se výčet funkcí zúží především na funkci *mmap*, která je základním prostředkem pro správu virtuální paměti v systému Linux potažmo Unix. Protože jsem do výčtu funkcí u systému Windows zařadil i získání velikosti alokovaného objektu z heapu, pro pořádek obdobnou funkci uvedu i v tomto textu. Tato funkce není ve standardu jazyka C, ale je řešena jako rozšíření v její standardní knihovně a jedná se o funkcí *malloc_usable_size*. Nicméně je nutné brát na zřetel, že tato funkce vrací skutečnou velikost, nikoli tu, která byla použita při alokaci.

Funkce pro správu virtuální paměti	
mmap	Tato funkce umožňuje mapovat zařízení do virtuální paměti. Pokud je použit deskriptor souboru, funguje jako mapování souborů do paměti. Je-li tento deskriptor ponechán nulový, funkce přidělí požadovaný počet stránek ve virtuální paměti. Filozofie této funkce spočívá ve faktu, že s virtuální pamětí je vždy svázán odkládací soubor a ten buď stanoví uživatel, nebo je automaticky přiřazen systémem z oddílu swap.
mremap	Na rozdíl od systému Windows umožňuje systém Linux změnu velikosti mapované oblasti. Díky tomu, že na systémové úrovni stačí pouze přemapovat virtuální adresy, což znamená opravit několik tabulek a změnit adresář stránek bez toho, aby se musel obsah dané oblasti kopírovat na jiné místo, lze pomocí této funkce implementovat velice efektivní realloc.
munmap	Slouží k odmapování úseku stránek z virtuální paměti. Toto odmapování lze provést na libovolném místě, kde byly stránky mapovány. Díky tomu je možné dočasně odmapovat stránky i uprostřed úseku.
mprotect	Obdobně jako na systému Linux lze v průběhu měnit nebo zjišťovat přístupová práva k jednotlivým stránkám. To může sloužit k ochraně určité oblasti paměti nebo například možnost spustit vygenerovaný program z jiné než kódové sekce. Bez tohoto mechanismu se například neobejde JIT kompilátor.