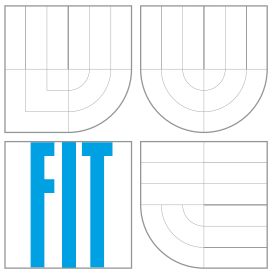
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO PRÁCI S OBJEKTY VE SDÍLENÉ PAMĚTI

LIBRARY FOR WORK WITH OBJECTS IN SHARED MEMORY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — KAMIL DUDKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL ŠPANĚL
SUPERVISOR

BRNO 2007

Knihovna pro práci s objekty ve sdílené paměti

Zadání

1. Prostudujte dostupné materiály na téma sdílená paměť a alokace objektů v C++.
2. Analyzujte a porovnejte současné knihovny pro práci s objekty ve sdílené paměti.
3. Vyberte vhodné přístupy a navrhněte jednoduchou knihovnu a demonstrujte její možnosti.
4. Experimentujte s vaší implementací a případně navrhněte vlastní modifikace metod.
5. Porovnejte dosažené výsledky a diskutujte možnosti budoucího vývoje. Zvažte další pokračování v rámci diplomové práce.
6. Vytvořte stručný plakát prezentující vaši bakalářskou práci, její cíle a výsledky.

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Tato bakalářská práce se zabývá problematikou meziprocesové komunikace v moderních operačních systémech. Důraz je kladen na využití sdílené paměti pro meziprocesovou komunikaci v objektově orientovaném jazyku C++. Součástí práce je popis návrhu a implementace knihovny, která umožňuje sdílenou paměť jednoduše a efektivně používat. Knihovna umožňuje sdílet přímo objekty jazyka C++ mezi procesy. Kromě toho vytváří knihovna platformově nezávislé rozhraní pro práci se sdílenou pamětí v operačních systémech Linux a Microsoft Windows. V závěru práce je zhodnocena efektivita využití sdílené paměti jako způsobu meziprocesové komunikace.

Klíčová slova

sdílená paměť, C++, knihovna, ICP, Linux, Windows, MDSTk, STL, alokátor, Share

Abstract

This bachelor's thesis considers problems of IPC (Inter Process Comunication) in modern operating systems. It is concentrated on usage of shared memory as IPC in object-oriented language C++. Thesis includes design and implementation of library, which provides easy end effective usage of shared memory. The library makes possible to share C++ language's objects between processes. Furthermore it creates platform-independent interface for work with shared memory on operating systems Linux and Microsoft Windows. Effectivity of shared memory usage as kind of ICP is evaluated in the conclusion of the thesis.

Keywords

shared memory, C++, library, ICP, Linux, Windows, MDSTk, STL, allocator, Share

Citace

Kamil Dudka: Knihovna pro práci s objekty ve sdílené paměti, bakalářská práce, Brno, FIT VUT v Brně, 2007

Knihovna pro práci s objekty ve sdílené paměti

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Michala Španěla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Kamil Dudka

12. května 2007

Poděkování

Děkuji svému vedoucímu Ing. Michalu Španělovi za odbornou pomoc při vývoji knihovny a v neposlední řadě také za poskytnutí zdrojových kódů toolkitu MDSTk a za souhlas s jejich využitím v této práci.

Obsah

Kapitola 1

Úvod

Sdílená paměť představuje efektivní nástroj pro meziprocesovou komunikaci, který je dostupný ve všech moderních operačních systémech. Využití nalézá při zpracování větších bloků dat (např. objemová obrazová data). V současnosti však neexistují efektivní a jednoduché nástroje pro její využití v C++. Předmětem této práce je návrh a implementace knihovny, která tento nedostatek řeší.

Kapitola 2 obsahuje informace o meziprocesorové komunikaci v obecné rovině, jsou zde vysvětleny základní technické pojmy, se kterými návrh knihovny pracuje. Kapitola 3 popisuje současný stav operačních systémů, překladačů a dostupných knihoven z hlediska práce se sdílenou pamětí. V kapitole 4 je vysvětlen návrh knihovny v několika krocích – zejména jsou zde popsány a zdůvodněny jednotlivé části rozhraní knihovny. Kapitola 5 shrnuje implementaci knihovny a popisuje některé zajímavé implementační detaily. Experimenty s knihovnou a jejich výsledky jsou uvedeny v kapitole 6.

Neodmyslitelnou pomůckou při práci s knihovnou je dokumentace knihovního API (*Application Programming Interface*), která se nachází na přiloženém CD. V příloze A se nachází ukázka zrojového kódu, který s knihovnou pracuje.

Kapitola 2

Teoretická část

V této kapitole jsou vysvětleny problémy, které vznikají při komunikaci procesů, a jejich možná řešení. Dále je představena sdílená paměť jako prostředek meziprocesorové komunikace, jsou zmíněny její výhody a nevýhody. Jde však pouze o přehled a seznámení čtenáře s použitou terminologií. Přesné definice jednotlivých pojmů je možné najít v odkazované literatuře.

2.1 Správa procesů v moderních OS

Proces je abstrakce definovaná operačním systémem (OS), která představuje program zavedený do paměti. Operační systém poskytuje procesu prostředky (procesor, paměť, ...) k tomu, aby mohl vykonávat potřebnou práci. Moderní OS umožňují spouštět více procesů současně, podporují tzv. multi-tasking. Aby mohly procesy běžet na jednom stroji nezávisle na sobě, snaží se je operační systém co nejvíc oddělit od sebe. Pokud například jeden proces havaruje, nechceme, aby jeho havárie negativně ovlivnila ostatní procesy běžící na stroji.

Správu procesů zajišťuje jádro operačního systému, zejména jeho části *plánovač* a *správce virtuální paměti*. Tyto mechanismy jsou pro proces zcela transparentní. Proces samotný se chová, jako by měl celý procesor pro sebe. Stejně tak ve svém adresovém prostoru nemůže číst data ostatních procesů běžících na stroji.

Na jednom procesoru může v jednom čase běžet pouze jeden proces. Procesorů máme však omezený počet, často jich máme méně než současně běžících procesů. Procesy potom běží *pseudoparalelně* – dochází k jejich přepínání na jednom procesoru. Spolu s přepínáním procesů se musí přepínat *kontext procesu*. Kontext procesu představuje stavové informace procesoru (nejčastěji množinu registrů procesoru, která je přístupná uživatelskému procesu[1]). Přepínání kontextu zajišťuje plánovač. Více o činnosti plánovače si můžete přečíst např. v [5].

[1] Kromě těchto registrů obsahují procesory také registry, které jsou dostupné pouze v privilegovaném režimu. O obsah těchto registrů se stará výhradně OS. Více informací najdete např. v [1]

2.2 Větvení procesů

Pokud v jednom programu potřebujeme využívat funkcionalitu jiného programu, máme dvě možnosti, jak to udělat:

- Budeme volat program nebo jeho část v rámci jednoho procesu.
- Požádáme OS o vytvoření nového procesu a v něm program spustíme.

Obě varianty mají své výhody i nevýhody. Výhoda první varianty spočívá v její jednoduchosti. Volající program se pozastaví na dobu běhu volaného programu. Nemusíme se tedy zabývat synchronizací procesů – to je dáno už tím, že máme ve skutečnosti proces jenom jeden. Oba programy budou také sdílet stejný adresový prostor.

Pokud vytvoříme nový proces, poběží oba programy paralelně[2]. Každý proces bude mít svůj oddělený adresový prostor. Havárie jednoho procesu nemusí nutně znamenat havárii druhého. Tento způsob vyžaduje nějakou práci navíc při vývoji aplikace a také nějakou režii při běhu. Moderní aplikace se však bez větvení procesů neobejdou.

Takový přístup se často používá např. u GUI aplikací, které umožňují komunikovat s uživatelem během provádění časově náročné operace.

2.3 Meziprocesorová komunikace

Jak bylo uvedeno v kapitole 2.1, procesy jsou od sebe zcela oddělené. Každý proces pracuje se svým kontextem a se svým adresovým prostorem. Přesto však někdy potřebujeme, aby spolu procesy komunikovaly – pracovaly nad společnými daty, nebo aby jeden proces informoval druhý o průběhu výpočtu, apod.

Operační systémy umožňují otevřít jeden soubor více procesy – je tedy možné komunikovat přes běžný soubor nacházející se někde na disku. Možnosti práce se souborem jsou však omezené, navíc některé OS mají problémy se sdílením souborů více procesy.

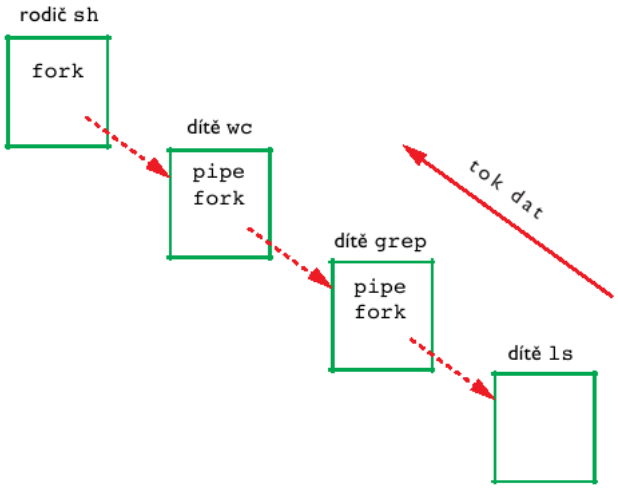
Naštěstí soubor na disku není jediný prostředek, pomocí kterého mohou procesy komunikovat. OS nabízí k tomuto účelu celou řadu mechanismů. Souhrnně se tyto mechanismy nazývají *IPC (Inter Process Comunication)*. Následuje přehled těch nejpoužívanějších:

- roury – pojmenované/anonymní
- signály
- zprávy
- sockety
- Remote Procedure Call (RPC)

[2]Na stroji s jedním procesorem poběží pseudoparalelně, na stroji s více procesory mohou běžet skutečně paralelně.

- Sdílená paměť

Ke každému způsobu komunikace je možné najít celou řadu příkladů použití. Některé druhy IPC jsou specifické pro konkrétní OS. Například signály se používají na Linuxu pro synchronizaci a plánování procesů. Zprávy používá systém Microsoft Windows pro komunikaci mezi GUI aplikacemi. Sockety a RPC jsou zase typické pro aplikace orientované na komunikaci přes síť. Na obrázku 2.1 je znázorněno použití anonymní roury, nevýhoda této metody spočívá v sekvenčním přístupu k datům.



Obrázek 2.1: Použití roury: Kolona procesů `ls | grep .c | wc` – obrázek převzatý z [10]

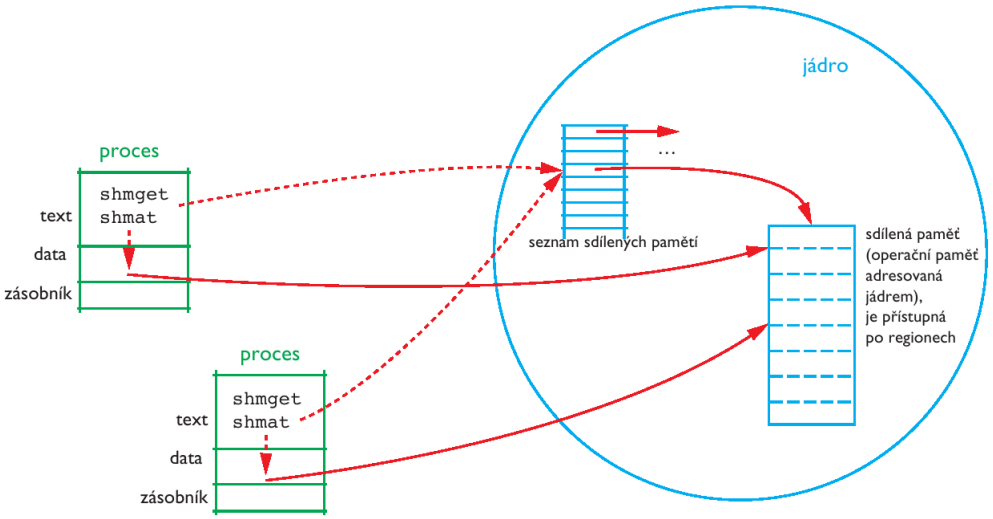
2.4 Sdílená paměť

Při studiu sdílené paměti jsem čerpal z [6]. Sdílená paměť je jedna z nejjednodušších IPC metod. Umožňuje dvěma nebo více procesům sdílet stejný kus paměti. Když jeden proces změní obsah paměti, všechny ostatní procesy změnu vidí.

Před použitím sdílené paměti musí nějaký proces alokovat segment sdílené paměti. Každý proces, který chce využívat přístup ke sdílenému segmentu, si jej potom musí připojit. Připojení spočívá v namapování sdíleného segmentu do adresového prostoru procesu[3]. Po ukončení práce se sdíleným segmentem je potřeba, aby se každý proces odpojil od sdíleného segmentu. Nakonec musí být sdílený segment uvolněn.

Přístup ke sdílené paměti je stejně rychlý jako přístup k nesdílené paměti procesu. Jakmile je sdílená paměť namapována do adresového prostoru procesu, není už dále potřeba volat služby OS pro přístup k paměti. Tím se můžeme vyhnout nadbytečnému kopírování dat z paměti do paměti (nebo na jiné úložiště).

[3]Sdílený segment se může v jednotlivých procesech mapovat na různé adresy. To je jeden z hlavních problémů, které bylo potřeba vyřešit při návrhu knihovny.



Obrázek 2.2: Sdílená paměť – obrázek převzatý z [10]

Operační systém Microsoft Windows nabízí jako alternativu ke sdílené paměti *soubor mapovaný do paměti*[4]. Práce se souborem mapovaným do paměti je podobná jako, kdybychom pracovali se segmentem sdílené paměti. Nějaký proces musí vytvořit mapování procesu do paměti. Každý proces, který chce využívat přístup k mapovanému souboru, se musí k mapování připojit. Po ukončení práce s mapovaným souborem se proces odpojí. Nakonec je potřeba mapování souboru zrušit. Tento přístup je obecnější než sdílená paměť – kromě komunikace mezi procesy jej lze využít také pro přístup k běžným souborům operačního systému v rámci jednoho procesu.

Protože operační systém nesynchronizuje přístupy ke sdílené paměti, je potřeba provádět vlastní synchronizaci přístupu ke sdíleným datům mezi procesy. Například proces nesmí číst data ze sdílené paměti, která tam ještě nebyla zapsána. Stejně tak dva procesy nesmí psát do jednoho místa v paměti ve stejnou dobu. Operační systém nemá prostředky k tomu, aby tyto konflikty detekoval a může tak dojít k poškození sdílených dat. Použití *semaforu*, které tento problém řeší, je probráno v následující kapitole.

2.5 Semafory

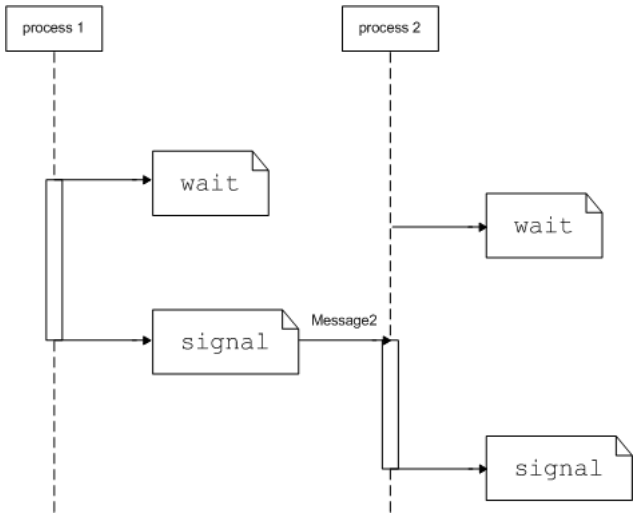
Popis práce se semafory v systému Linux najdete rovněž v [6]. Semafor je synchronizační prostředek poskytovaný operačním systémem. Zapouzdřuje čítač typu `int`, který má vždy kladnou hodnotu a poskytuje dvě základní operace:

- Operace `wait` sníží hodnotu čítače o 1. Pokud je již hodnota čítače nulová, proces je pozastaven dokud není hodnota opět kladná (v důsledku operace `signal` jiného procesu).

[4] Mapování souboru do paměti nabízí také Linux, v knihovně však byla použita sdílená paměť.

- Operace `signal` zvýší hodnotu čítače o 1. Pokud byla hodnota čítače nulová a jiný proces byl zablokován během volání `wait`, bude tímto voláním odblokován.

Operační systém zaručuje, že tyto operace jsou bezpečné vzhledem k současnému běhu procesů. Pomocí semaforů je tedy možné synchronizovat přístup k datům ve sdílené paměti, jak ukazuje obrázek 2.3. Semafory je potřeba alokovat a dealokovat podobně jako sdílené segmenty.



Obrázek 2.3: Sekvenční diagram běhu dvou procesů při použití semaforu

Semafory představují nízkoúrovňový synchronizační nástroj operačního systému. Pomocí semaforů lze řešit klasické synchronizační problémy, práce s nimi je však příliš složitá a proto se málokdy používají přímo. Existují různé abstrakce vybudované nad semafory, které řeší konkrétní situace. Synchronizačními problémy a jejich řešením se zabývá [5].

Monitor je abstraktní datový typ, který zapouzdřuje sdílená data. Nad sdílenými daty je možné v rámci monitoru implementovat určité operace, přičemž je zaručeno, že v jednom čase bude vždy rozpracována pouze jedna z těchto operací.

2.6 Virtuální metody

Jazyk C++ má, stejně jako většina objektově orientovaných jazyků, podporu pro *dynamický polymorfismus*[5]. Definice polymorfismu podle Wikipedie[13]:

> Odkazovaný objekt se chová podle toho, jaký je jeho skutečný typ. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší. V praxi se tato vlastnost projevuje např. tak,

[5]Kromě dynamického polymorfismu disponuje jazyk C++ také statickým polymorfismem ve formě šablon.

že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy (třídy, která přímo či nepřímo z této třídy dědí), která se může chovat jinak, než by se chovala instance rodičovské třídy, ovšem v rámci mantinelů, daných popisem rozhraní.

Dynamický polymorfismus je v C++ realizován pomocí *virtuálních metod*. Virtuální metody jsou metody volané nepřímo – před samotným voláním se provede výběr správné metody na základě typu objektu, pro který je metoda volána. Úvod do virtuálních metod lze nalézt v [8], podrobnější informací pak v [11].

Citace z [7]:

> Pro pochopení virtuálních metod je vhodné vědět, jak je uvedený mechanismus obvykle implementován:
>
> - Každá třída s virtuálními metodami má tzv. tabulku virtuálních metod (VMT - Virtual Method Table), ve které jsou odkazy na všechny virtuální metody třídy.
> - Při dědění (při překladu) se převezme obsah VMT bázové třídy, odkazy na virtuální metody, které byly předefinovány se nahradí novými a na konec VMT se doplní odkazy na případné nové virtuální metody.
> - Každý objekt třídy s virtuálními metodami obsahuje odkaz na tabulku virtuálních metod.
> - Polymorfní volání použije ukazatel v objektu, vybere odpovídající položku z VMT a zavolá metodu.

Kapitola 3

Současný stav

Dnes dostupné překladače jazyka C++ nemají vestavěnou podporu pro práci se sdílenou pamětí. Jsme tedy odkázáni na API operačního systému, pomocí kterého je sdílená paměť zpřístupňována. Stejně tak ve standardu C++ nejsou zahrnuté prostředky pro synchronizaci procesů. Některé překladače umožňují např. definovat kritickou sekci, ale tato řešení jsou platformově závislá. Neexistuje jednotné API pro práci se sdílenou pamětí a synchronizačními prostředky. Pokud tedy píšeme multiplatformní aplikaci, nezbývá než implementovat práci se sdílenou pamětí na každém OS jinak.

V důsledku nízké podpory překladače je sdílení objektů jazyka C++ velmi omezené. Překladače splňující nějaký standard jazyka C++ se zavazují implementovat syntaxi a sémantiku jazyka, kterou standard předepisuje. Samotný překlad do binárního kódu však standardizován není. Budeme-li definovat ve dvou různých překladačích objekt stejným způsobem, není zaručeno, že bude v paměti stejně uložen.

Při používání objektů jazyka C++ jako komunikačního prostředku, se tedy může projevit binární nekompatibilita mezi komunikujícími programy. Existuje podmnožina objektů, zvaných POD, pro které je binární kompatibilita zaručena. Zkratka POD znamená *Plain Old Data*, definici můžete nalézt v [3]. Jednoduše řečeno, jedná se o data, která lze kopírovat bit po bitu. Pokud chceme mít zajištěnou binární kompatibilitu mezi aplikacemi, které komunikují přes sdílenou paměť, měli bychom komunikovat pomocí objektů POD[1].

Součástí standardu C++ je také STL (*Standard Template Library*). Jedná se o knihovnu šablon implementující běžně používané typy kontejnerů pro objekty (zásobník, vektor, ...) a algoritmů, které s nimi umí pracovat (řazení, vyhledávání, ...). Úvod do práce s STL naleznete např. v [11]. Návrh STL realizuje nezávislost kontejnerů na fyzickém umístění dat pomocí tzv. *alokátorů*. Implicitní alokátor používá run-time jazyka C++ a jeho správu dynamicky alokované paměti. Můžeme si však vytvořit vlastní alokátor, který bude ukládat data do sdílené paměti[2]. Způsob vytvoření alokátoru a jeho využití při práci s STL je

[1] Typový systém jazyka C (předchůdce C++) je založený pouze na objektech POD.

[2] Alokátor objektů využívající sdílenou paměť představuje jeden typ alokátoru. Je možné vytvořit také alokátor perzistentních objektů uložených na disku apod.

popsáno rovněž v [11]. Bohužel některé implementace STL nepracují s alokátory tak, jak je uvedeno v této knize.

V současné době existuje otevřená implementace alokátoru pro STL[3], který pracuje se sdílenou pamětí. Podrobnosti o návrhu alokátoru lze nalézt v [9]. Tento projekt byl však zastaven, podle autora, z důvodu bugu[4] v překladači GCC. Vývojáři překladače však tento bug označili za neplatný. Od té doby se projekt dále nevyvíjí. Navíc podle dostupných informací alokátor nefunguje na současné stabilní verzi překladače GCC.

Implementace platformově závislé části knihovny vychází z toolkitu MDSTk[14] pro segmentaci medicínských dat. Tento toolkit obsahuje mimo jiné knihovnu, která umožňuje práci se sdílenou pamětí a semafory na systémech Linux a Microsoft Windows. Tyto prostředky operačního systému jsou na obou platformách zpřístupňovány se stejným rozhraním. Tím je vytvářena transparentní vrstva pro aplikace, které knihovnu používají.

[3] `http://allocator.sourceforge.net/`

[4] Podrobnosti zaslaném bugu naleznete na `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=21251`

Kapitola 4

Návrh řešení

Při návrhu bylo hlavním cílem knihovny zjednodušit práci s objekty ve sdílené paměti. Každý programátor C++ umí pracovat s objekty jazyka C++. Bylo tedy potřeba práci s objekty ve sdílené paměti maximálně připodobnit práci s nesdílenými objekty.

4.1 Alokace sdílených objektů

V jazyce C++ existují dva způsoby, jak vytvořit instanci třídy – *objekt*:

1. **Staticky** – Data objektu jsou uložena přímo na zásobníku běžícího procesu. Konstrukci a destrukci objektu řídí překladač automaticky na základě vstupu a výstupu z bloku, ve kterém je objekt definován.

   ```
   MyClass myObject(...);
   ```

2. **Dynamicky** – Data objektu jsou uložena na hromadě. Konstrukci a destrukci řídí programátor voláním `new` a `delete`.

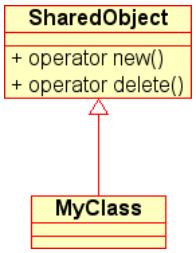
   ```
   MyClass *myObject = new MyClass(...);
   ...
   delete myObject;
   ```

Je zřejmé, že statické objekty nelze sdílet – umístění objektů na zásobník zajišťuje překladač již v době překladu[1]. Navíc nemáme zcela pod kontrolou dobu existence objektů. Příznivější (z hlediska sdílení) je situace u dynamicky alokovaných objektů. Stačí místo volání `new` a `delete` volat odpovídající metody v rozhraní knihovny a tím získat kontrolu nad životem objektů.

[1] Jazyk C již od verze ISO C99 umožňuje za běhu stanovit rozměr pole uloženého na zásobníku. Z pohledu objektového jazyka C++ však tato možnost není tolik přínosná.

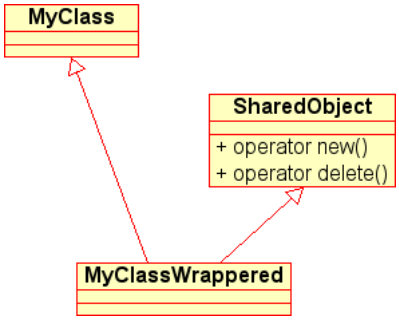
Uživatel knihovny by však nebyl příliš nadšený, kdyby musel zasahovat do stávajícího kódu a zaměňovat veškeré operátory `new` a `delete` za volání nějakých metod. Jazyk C++ však s touto situací počítá a umožňuje operátory `new` a `delete` **přetížit**. Pro stávající kód pracující s objektem se tak změna úložiště stává zcela transparentní – v kódu jsou nadále použity operátory `new` a `delete`, mají však odlišnou implementaci.

Tím je vyřešen problém s vytvořením instance sdíleného objektu. Nyní je třeba zvolit vhodnou strategii pro tvoření samotných tříd, jejichž objekty se budou sdílet. Při návrhu jsem vycházel ze zásady „Skrývejte informace“ z [12]. Přetížení operátorů jsem zapouzdřil do samostatné třídy, kterou jsem nazval `SharedObject`. Když bude uživatel knihovny deklarovat třídu umožňující sdílení, stačí použít **jednoduchou dědičnost**, jak ukazuje obr. 4.1. Pokud je třída odvozena z třídy `SharedObject`, neznamená to samozřejmě, že není možné objekty této třídy alokovat na zásobníku nebo na hromadě. Pokud není připojen sdílený segment, chování objektu se nijak neliší od původního (viz. dále).



Obrázek 4.1: Deklarace nové třídy umožňující sdílení.

Můžeme se však dostat do situace, že máme třídu již hotovou a chceme pro ni zajistit sdílení tak, abychom ji nemuseli měnit. I v tomto případě je možné použít třídu `SharedObject` – tentokrát pomocí **vícenásobné dědičnosti**. Již existující třídu je třeba spolu se třídou `SharedObject` obalit novou třídou, jak je znázorněno na obr. 4.2. Na první pohled by se mohlo zdát, že se tím zvětší velikost alokovaných objektů. Třída `SharedObject` má však nulovou velikost, protože neobsahuje členská data. Odvozená třída tedy bude mít stejnou velikost, jako třída původní.



Obrázek 4.2: Obalení existující třídy.

Odpovídající kus kódu v jazyce C++ bude vypadat takto:

```
class MyClassWrappered:
  public MyClass,
  public SharedObject
  { };
```

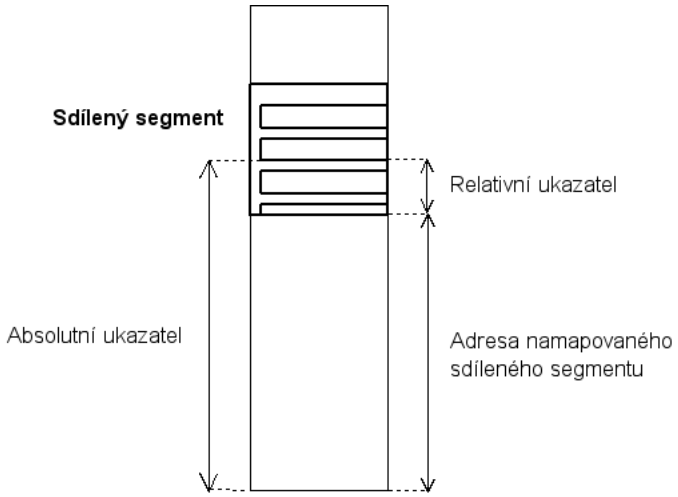
4.2 Ukazatel na sdílený objekt

V předchozí kapitole byl vyřešený problém s alokací objektů uvnitř sdíleného segmentu v rámci jednoho procesu. To však samo o sobě nepřináší žádný užitek. Alokované objekty je potřeba sdílet mezi procesy.

Při úspěšném volání `new` v uvedeném příkladu bude ukazatel `myObject` ukazovat na nově alokovaný objekt v připojeném sdíleném segmentu. Pomocí tohoto ukazatele můžeme s objektem pracovat uvnitř procesu, který objekt alokoval. Problém nastane, když se snažíme ukazatel předat procesu, se kterým komunikujeme.

Jak bylo řečeno v kapitole 2.1, každý proces pracuje se svým paměťovým prostorem, o který se stará správce virtuální paměti. Sdílený segment se může v jednotlivých procesech mapovat na různé adresy. Ukazatel na objekt, který byl získán při alokaci objektu v jednom procesu, může být v kontextu jiného procesu neplatný.

Ukazatel tedy nelze jednoduše sdílet mezi více procesy – je potřeba zohlednit adresu, na které je sdílený objekt v každém procesu namapovaný. Ukazatel lze pro tento účel rozdělit na dvě části, jak je znázorněno na obr. 4.3. První část je adresa namapovaného



Obrázek 4.3: Ukazatel na sdílený objekt v adresovém prostoru procesu

sdíleného segmentu. Druhá část představuje umístění objektu vzhledem k počátku sdíleného segmentu. Je zřejmé, že druhá část ukazatele bude pro všechny připojené procesy stejná.

Pro účely návrhu knihovny jsem tuto část nazval *relativní ukazatel*. Celý ukazatel (vrácený voláním `new`) jsem pak pro rozlišení označil jako *absolutní ukazatel*.

Komunikující procesy si mohou mezi sebou předávat relativní ukazatele na objekty. Pokud však chtějí s objektem pracovat, musí použít absolutní ukazatel. Při komunikaci je proto potřeba provádět *překlad adres* – převádět relativní ukazatele na absolutní a naopak. Překlad adres by měl být automatický a pokud možno skrytý před uživatelem knihovny. Za tímto účelem jsem definoval vlastní typ ukazatele na sdílený objekt – `RelocPtr`[2].

Ukazatele tohoto typu lze bez problémů sdílet mezi procesy, uvnitř je totiž uložena pouze relativní adresa objektu. Navenek se však tváří jako běžný ukazatel, tj. absolutní. Toho je dosaženo přetížením operátorů `->`, `*`, `[]` a operátorů ukazatelové aritmetiky. Aby bylo možné ukazatele samotné umisťovat do sdíleného segmentu voláním `new`, je třída `RelocPtr` odvozena ze třídy `SharedObject` uvedené v předchozí kapitole.

Ukazatel `RelocPtr` nemá sám o sobě dost informací k výpočtu absolutního ukazatele, výpočet proto neprovádí přímo metody třídy `RelocPtr`. K tomuto účelu je v každém procesu instance třídy `ShareManager`, která mimo jiné zajišťuje připojení sdíleného segmentu. Má tedy k dispozici potřebné informace pro překlad adres.

4.3 Rozhraní knihovny

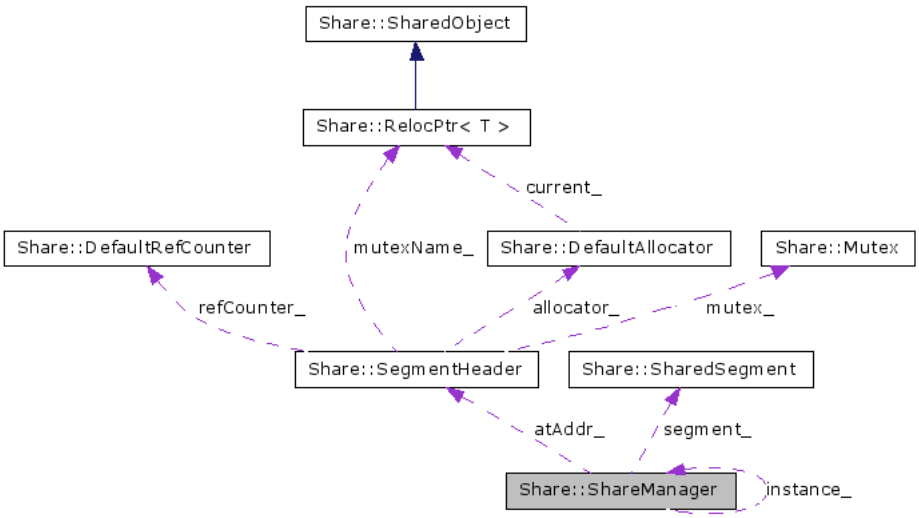
V přiloženém CD se nachází dokumentace knihovního API (Application Programming Interface). V této dokumentaci lze najít kompletní popis rozhraní knihovny. Dokumentace je rozdělena do několika částí podle účelu dokumentovaných tříd.

Hlavní část rozhraní knihovny tvoří třída `ShareManager`. Při jejím návrhu byl použit návrhový vzor *Singleton*, který je podrobně popsán v [4]. Tento návrhový vzor zajistí, aby byla v jednom procesu vytvořena vždy maximálně jedna instance této třídy a byla dostupná ze všech částí zdrojového kódu. Třída `ShareManager` umožňuje:

- Vytvořit nový sdílený segment
- Připojit/odpojit již existující sdílený segment
- Alokovat bloky paměti uvnitř sdíleného segmentu
- Převádět relativní ukazatele na absolutní a naopak

K provedení těchto operací třída `ShareManager` využívá třídy nižší vrstvy knihovny, jak je uvedeno na obr. 4.4. Objekt třídy `SegmentHeader` je umístěn na začátku každého sdíleného segmentu, je tedy sdílen všemi připojenými procesy. Třída `SharedSegment` zapouzdřuje připojený sdílený segment a tvoří platformově nezávislou vrstvu pro práci se sdílenou pamětí. Implementace těchto tříd je popsána v kapitole 5.

[2] Přesněji řečeno, jedná se o šablonu `RelocPtr<T>`, kde `T` je typ sdíleného objektu.



Obrázek 4.4: Diagram spolupráce třídy `ShareManager`

Součástí knihovního rozhraní je také šablona pro STL alokátor – `Allocator<T>`. Pomocí této šablony je možné sdílet kontejnery STL. V kapitole 5 je ukázáno, jak se tato šablona používá.

4.4 Omezení při práci s knihovnou

V této kapitole jsou popsána a zdůvodněna omezení, která jsou kladena na programy pracující s knihovnou.

Virtuální metody

Hlavní omezení kladené na programy, které knihovnu využívají, se týká virtuálních metod (viz. kapitola 2.6). Tak jako nelze sdílet absolutní ukazatel na sdílený objekt mezi procesy, nelze sdílet ani ukazatel na tabulku virtuálních metod.

Tento ukazatel se nastavuje při vytváření objektu – je tedy platný v kontextu procesu, který jej vytvořil. Ukazatel na tabulku virtuálních metod se však sdílí spolu s členskými daty a v kontextu jiného procesu může být neplatný. Pokud zavoláme virtuální metodu sdíleného objektu z jiného procesu, může program havarovat.

Manipulace s ukazatelem na tabulku virtuálních metod je řízena výhradně překladačem. Není tedy v silách knihovny tento problém nějak řešit. Při použití knihovny je proto nutné s tímto omezením počítat – **sdílené objekty nesmí obsahovat žádné virtuální funkce**.

Počet současně připojených segmentů

O připojení/odpojení sdíleného segmentu se stará samotný singleton `ShareManager`,

přičemž připojený segment není nijak identifikován. Z toho vyplývá, že **nelze připojit najednou více sdílených segmentů**.

Příčinou samozřejmě není existence singletonu. Byla zvažována také varianta návrhu pracující s více sdílenými segmenty. Problém je však s výkonem. Nejslabším místem knihovny po stránce výkonu je překlad adres. V současné verzi knihovny spočívá překlad adres pouze v přičtení adresy připojeného segmentu k relativnímu ukazateli[3].

Při práci s více segmenty by bylo vždy potřeba nastavit aktivní segment. Při každém překladu adres by potom bylo nutné zjistit, který segment je v danou chvíli aktivní. To je časově velmi náročná operace vzhledem k četnosti překladu adres. Pokud by navíc programátor zapomněl nastavit aktivní segment, překlad adres by se provedl špatně – tím by se zvýšilo riziko chyby v programech, které knihovnu používají.

Z těchto důvodů byla zvolena varianta návrhu umožňující připojit pouze jeden sdílený segment a přijato tohle omezení.

[3]Ikdyž může graf volání při překladu adres vypadat hrozivě, většina funkcí je `inline` – to znamená, že se funkce rozbalí při příkladu místo použití volací konvence. Výsledek potom v binární podobě vypadá mnohem jednodušeji.

Kapitola 5

Implementace

Z hlediska implementace se třídy knihovny dělí do následujících částí:

- Třídy jádra knihovny
- Odlehčená implementace některých STL kontejnerů
- Utilita `sharectl`
- Interní třídy knihovny

Implementační detaily naleznete v dokumentaci knihovního API na přiloženém CD. Kromě popisu jednotlivých tříd obsahuje dokumentace také odkazy přímo do zdrojového kódu knihovny. Následuje stručný popis jednotlivých částí.

Třídy jádra knihovny

Do této části spadají třídy tvořící rozhraní knihovny, které byly uvedeny v kapitole 4. Knihovna dále definuje vlastní typ výjimky – `Share::Exception`. Tato výjimka je odvozena od `std::bad_alloc`, navíc však obsahuje textový řetězec, který popisuje chybu. Tento řetězec může být využit pro účely ladění.

Z implementačního hlediska jsou zajímavé šablony `TReference<T>` a `TVoidPointer<T>`. Jedná se o tzv. *rysy* (anglicky *traits*). Šablona `TReference<T>` zabraňuje vytvoření reference na `void` v šablonách, které ji používají. Šablona `TVoidPointer<T>` definuje typ ukazatele na `void`, přičemž ponechá konstantní ukazatel konstantním. Informace o tom, jak rysy fungují, naleznete v [2].

Strukturu `Allocator::rebind<U>` používají STL kontejnery k vytvoření jiného typu alokátoru na základě existujícího typu. Proč je tohle potřeba, vysvětluje [11]. Názorný příklad použití naleznete v implementaci kontejneru `Share::Map`.

Odlehčená implementace některých STL kontejnerů

Součástí knihovny je odlehčená implementace některých STL kontejnerů. Tyto kontejnery si nekladou za cíl nahradit stávající implementace STL. Pouze demonstrují možnosti jádra knihovny a ukazují, že je možné kontejnery sdílet mezi procesy[1].

Tyto kontejnery implementují pouze podmnožinu operací běžných STL kontejnerů. Jejich rozhraní se může lišit v některých detailech. Jsou však názorně použity v testovacích příkladech. V tabulce 5.1 se nachází přehled implementovaných kontejnerů.

STL	Share
`std::vector`	`Share::Vector`
`std::string`	`Share::String`
`std::map`	`Share::Map`

Tabulka 5.1: Názvy implementovaných STL kontejnerů

Kontejnery používají abstrakci alokátoru, stejně jako standardní kontejnery. Jako výchozí alokátor je nastaven `std::allocator`. Pokud jsou použity kontejnery s tímto alokátorem, jsou nezávislé na práci se sdílenou pamětí.

Pokud chceme kontejnery sdílet, je nutné nastavit jako alokátor `Share::Alocator`. Následující ukázka (převzata z testovacího programu `test2_wordcount`) ukazuje, jak uvnitř sdíleného objektu vytvořit mapování z řetězce na číslo:

```
class WordCount: public Share::SharedObject {
  typedef Share::String <Share::Allocator <char> > TString;
  typedef Share::Allocator <Share::Pair <TString, int> > TAllocator;
  typedef Share::Map<TString, int, TAllocator> THashTable;

  THashTable hashTable_;
  ...
};
```

Utilita `sharectl`

Spolu s knihovnou je také dodávána konzolová aplikace `sharectl`, která s knihovnou spolupracuje. Tato aplikace umožňuje sledovat sdílené segmenty. To se hodí při ladění aplikací, které pracují s knihovnou, a také při ladění knihovny samotné.

Kromě výpisu základních statistik sdíleného segmentu dokáže tato utilita například odstranit sdílený segment, který zůstane v paměti po havárii aplikace. Také lze touto

[1] Běžně používané implementace STL při testech nepracovali s knihovním alokátorem správně, více v kapitole 6.

utilitou ručně volat operace nad sdíleným segmentem, jako je alokace bloku paměti, zvýšení počtu počítadla referencí, apod.

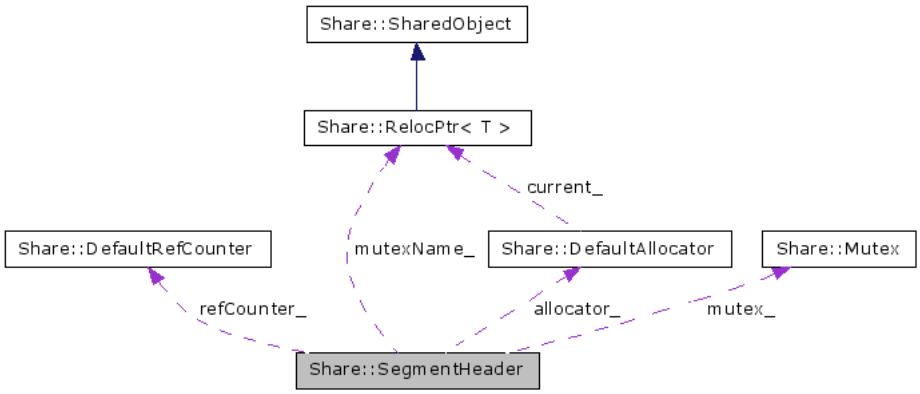
Nápovědu k `sharectl` lze nalézt v programové dokumentaci, nebo pomocí příkazu:

```
sharectl --help
```

Interní třídy knihovny

Do této části spadají třídy, kterou jsou uživateli knihovny skryty. Nejnižší vrstvu knihovny tvoří třídy zajišťující platformovou nezávislost zbytku knihovny – `SharedSegment` a `Mutex`. Při implementaci těchto tříd byly použity zdrojové kódy MDSTk[14].

Sdílená data knihovny pro každý segment jsou uložena v objektu `SegmentHeader`, který je umístěn na začátku sdíleného segmentu. Tento objekt spojuje funkcionalitu alokátoru a počítadla referencí. Sám se navíc chová jako monitor (viz. kapitola 2.5), čímž zajišťuje bezpečnost alokace a počítání referencí při paralelním běhu procesů.



Obrázek 5.1: Diagram spolupráce třídy `SegmentHeader`

Vzájemné vyloučení procesů je zajištěno třídou `Mutex`, graf spolupráce je na obr 5.1. Třídy `DefaultAllocator` a `DefaultRefCounter` lze nahradit jinými (dokonalejšími) třídami změnou definice typů `TSegmentAllocator` a `TSegmentRefCounter` v souboru `SegmentHeader.h`.

Kapitola 6

Výsledky

Předmětem této práce je knihovna. Knihovna samotná se však testuje velmi špatně. Aby bylo možné vyzkoušet funkci knihovny a zhodnotit její vlastnosti, napsal jsem sadu testovacích programů. Tyto programy jsou distribuovány spolu s knihovnou.

6.1 Testovací programy

Základním diagnostickým nástrojem při práci s knihovnou je utilita `sharectl`, která byla uvedena v kapitole 5. Na rozdíl od ostatních programů a testovacích skriptů pracuje s interními třídami knihovny – slouží tedy k diagnostice knihovny na nejnižší úrovni.

Pomocí této utility lze ladit ostatní testovací programy, ale také knihovnu samotnou. Umožňuje například zjistit nekompatibilitu s operačním systémem nebo jeho konfigurací. Pomáhá také překonat některá omezení OS, která jsou uvedena v následující kapitole. Dále jsou k dispozici tři jednoduché testovací programy, na kterých jsou funkce knihovny předvedeny:

`test1_string`

Tento jednoduchý testovací program představuje práci s knihovnou ve své nejjednodušší podobě. Na začátku je deklarována třída sdíleného objektu, který obsahuje řetězec[1]. Samotná funkce `main()` potom rozlišuje mezi dvěma režimy – proces se chová buď jako producent, nebo jako konzument.

Pokud je program spuštěn s jedním parametrem (jménem sdíleného segmentu), chová se jako producent. Vytvoří sdílený segment o velikosti `MAX_DATA_SIZE` a v něm alokuje sdílený objekt. Potom jsou načítány řádky ze standardního vstupu a ukládány do sdíleného objektu.

Pokud je program spuštěn s parametrem navíc (relativní adresou objektu), chová se jako konzument. Připojí se ke sdílenému segmentu, převede relativní adresu na ukaza-

[1] Implementace této třídy je velmi jednoduchá, bohužel však na úkor efektivity. Pokud potřebujete sdílet řetězec, použijte kontejner `Share::String`.

tel a vypíše obsah objektu na standardní výstup. Během komunikace není prováděna žádná synchronizace mezi procesy.

`test2_wordcount`

Program `test2_wordcount` jde v testování knihovny ještě dál. Uvnitř sdíleného objektu jsou sdíleny také STL kontejnery. Nejedná se o standardní STL kontejnery, ale o jejich knihovních ekvivalenty `Share::String` a `Share::Map`, které lépe pracují s alokátorem Share::Allocator.

Program, jak jeho název napovídá, slouží k počítání slov. Inicializace je provedena pří spuštění programu s jedním parametrem – názvem již alokovaného sdíleného segmentu. Po úspěšné inicializaci je vypsána relativní adresa alokovaného objektu, kterou je nutné programu předávat při jeho dalších voláních.

Pokud je jako třetí parametr zadáno jméno souboru, jsou do statistik zahrnuta slova v něm obsažená. Je-li program spuštěn pouze se dvěma parametry, jsou vypsány statistiky počtu slov. Datové struktury obsahující statistiky jsou uloženy ve sdíleném segmentu a tím jsou perzistentní vzhledem k jednotlivým voláním programu.

`test3_benchmark`

Tento testovací program, stejně jako jeho předchůdce, počítá slova. Na rozdíl od programu `test2_wordcount` však nezobrazuje statistiky počtu slov, ale dobu počítání slov. Program tedy slouží k měření výkonnostních charakteristik knihovny.

Během jednoho spuštění programu je postupně provedeno 24 různých testů. Popis jednotlivých testů spolu s naměřenými výsledky jsou uvedeny v kapitole 6.4. Během provádění jednotlivých testů nedochází k meziprocesorové komunikaci.

6.2 Testování knihovny na jednotlivých platformách

Linux (64bit)

Sestavení knihovny na operačním systému Linux je zajišťováno pomocí `GNU make`. Vývoj knihovny probíhal na distribuci *Gentoo Linux*. Překlad knihovny byl testován na překladačích `GCC 3.4.6` a `GCC 4.1.1`. Pomocí programu `Doxygen` je možné vygenerovat programovou dokumentaci.

Testovací programy pracují s velkými bloky sdílené paměti. Pro správnou funkci těchto programů je nutné nastavit odpovídající limity pro práci se sdílenou pamětí v jádru. To lze provést změnou obsahu následujících souborů v souborovém systému `proc`[2]:

[2]Pro zápis do těchto souborů je nutné mít odpovídající oprávnění.

```
/proc/sys/kernel/shmmni
/proc/sys/kernel/shmall
/proc/sys/kernel/shmmax
```

Implementace sdílené paměti na systému Linux umožňuje uchovat v paměti sdílený segment, ke kterému není připojený žádný proces. To je výhoda proti systému Microsoft Windows, který automaticky odstraňuje sdílené segmenty po ukončení posledního připojeného procesu.

Tohle chování však někdy může dělat problémy – segment zůstane „viset“ v paměti, když program havaruje. Takový segment je potom možné odstranit voláním:

```
sharectl --force-destroy SEGMENT_NAME
```

Microsoft Windows (32bit)

K sestavení knihovny na systému Microsoft Windows je potřeba mít nainstalované *Microsoft Visual Studio 2005* s podporou jazyka C++. Stačí otevřít soubor řešení `sharelib.sln` a kliknout na položku *Build solution* v menu *Build*. Tím se sestaví knihovna, utilita `sharectl` i testovací programy. Knihovna byla testována na *Windows XP Professional* a *Windows Vista Business*, které běžely na 32-bitovém virtuálním stroji *VMware Server*.

Jak bylo řečeno, Microsoft Windows automaticky odstraňují nepoužívané sdílené segmenty. To je většinou nežádoucí a v aplikacích, které s knihovnou pracují, je nutné s tímto chováním počítat. Pro účely ladění lze vytvořit sdílený segment pomocí příkazu:

```
sharectl --create-and-wait SEGMENT_NAME SEGMENT_SIZE
```

Utilita vytvoří sdílený segment s danými parametry a před návratem do příkazové řádky počká na stisk klávesy. Do stisknutí klávesy se potom chování sdíleného segmentu podobá chování na Linuxu.

6.3 STL std::string

Knihovní alokátor `Share::Allocator` byl navrhnut za účelem sdílení STL kontejnerů. Při testech však nefungoval podle plánu – STL kontejnery interně pracovaly s jiným typem ukazatele než definoval alokátor. To mělo za následek, že ke sdíleným datům nebylo možné přistupovat z více procesů.

Z tohoto důvodu disponuje knihovna vlastní odlehčenou implementací některých STL kontejnerů. Jejich srovnání se standardní implementací STL po stránce výkonu naleznete v následující kapitole.

Kromě toho jsem se pokusil upravit stávající implementaci STL tak, aby pracovala správně s knihovním alokátorem. Pro experiment jsem použil implementaci `std::string` v knihovně STL, která byla nainstalovaná na mé distribuci Linuxu. Navzdory její robustnostnosti byla samotná úprava knihovny velmi jednoduchá – stačilo změnit jediný řádek v hlavičkovém souboru `bits/basic_string.h`. Původní úsek změněného kódu vypadal takto:

```
struct _Alloc_hider : _Alloc
{
  _Alloc_hider(_CharT* __dat, const _Alloc& __a)
  : _Alloc(__a), _M_p(__dat) { }

  _CharT* _M_p; // The actual data
};
```

Změněný úsek kódu potom vypadal takto:

```
struct _Alloc_hider : _Alloc
{
  _Alloc_hider(_CharT* __dat, const _Alloc& __a)
  : _Alloc(__a), _M_p(__dat) { }

  pointer _M_p; // The actual data.
};
```

Po provedení této jednoduché úpravy bylo možné `std::string` sdílet mezi procesy a pracovat s ním stejně jako s `Share::String`. Standardní implementace je však oproti knihovní implementaci mnohem bohatější v počtu metod a spolupracujících algoritmů.

6.4 Výkon aplikací při práci s knihovnou

K měření výkonu při práci s knihovnou slouží testovací program `test3_benchmark`. Jádro programu vychází z třídy `WordCount` z programu `test2_wordcount`, tentokrát jde však o šablonu třídy. Parametry šablony jsou následující:

- Typ kontejneru použitý pro řetězec
- Typ kontejneru použitý pro mapování řetězce na číslo
- Typ alokátoru

Třída je pak postupně vytvářena s různými parametry šablony, testování je tak provedeno na osmi různých specializacích šablony `WordCount`. Kromě toho jsou objekty alokovány třemi možnými způsoby:

- Na zásobníku
- Na hromadě
- V připojeném segmentu sdílené paměti

Celkem je tedy provedeno 24 různých testů, u každého z nich je změřen čas potřebný pro spočítání slov. Výsledky testů pro jednotlivé platformy jsou v tabulkách 6.1 a 6.2. Jako vstupní soubor jsem použil databázi slov knihovny *CrackLib*. Celkově horší časy na platformě MS Windows jsou způsobeny mimo jiné tím, že testy běžely na virtuálním stroji.

Při testech 1-4 používal program standardní alokátor. Ikdyž byl objekt vytvořen ve sdílené paměti (poslední sloupeček tabulky), data kontejneru byla alokována na hromadě. Zajímavostí je, že při použití knihovní implementace řetězce byla rychlost **2x větší** než u `std::string`.

Plně funkční specializaci šablony `WordCount`, která byla použita v testovacím programu `test2_wordcount`, představuje test č. 8. Tato varianta používá výhradně třídy knihovny. Její rychlost je asi **3-5x nižší** než varianta s klasickou STL bez využití sdílené paměti (test č. 1).

Zajímavostí je také rozdíl mezi časy při statické a dynamické alokaci objektu `WordCount`. Tento rozdíl je pravděpodobně způsoben nižší lokalitou odkazů při alokaci objektu na zásobníku – část dat kontejneru se totiž nachází na zásobníku a část na hromadě.

Test č.	TString	TMap	TAllocator	Stack	Heap	SHM
1	std::string	std::map	std::allocator	0.24 s	0.14 s	0.14 s
2	Share::String	std::map	std::allocator	0.17 s	0.09 s	0.07 s
3	std::string	Share::Map	std::allocator	0.23 s	0.11 s	0.11 s
4	Share::String	Share::Map	std::allocator	0.21 s	0.1 s	0.09 s
5	std::string	std::map	Share::Allocator	0.35 s	0.17 s	0.39 s
6	Share::String	std::map	Share::Allocator	0.53 s	0.26 s	0.65 s
7	std::string	Share::Map	Share::Allocator	0.34 s	0.18 s	0.41 s
8	Share::String	Share::Map	Share::Allocator	0.44 s	0.24 s	0.64 s

Tabulka 6.1: Výsledky testovacího programu `test3_benchmark` na systému Linux.

Test č.	TString	TMap	TAllocator	Stack	Heap	SHM
1	std::string	std::map	std::allocator	1.063 s	0.219 s	0.203 s
2	Share::String	std::map	std::allocator	0.656 s	0.203 s	0.187 s
3	std::string	Share::Map	std::allocator	0.687 s	0.187 s	0.187 s
4	Share::String	Share::Map	std::allocator	0.453 s	0.219 s	0.219 s
5	std::string	std::map	Share::Allocator	1.062 s	0.266 s	9.953 s
6	Share::String	std::map	Share::Allocator	1.5 s	0.39 s	15.359 s
7	std::string	Share::Map	Share::Allocator	0.61 s	0.157 s	8.579 s
8	Share::String	Share::Map	Share::Allocator	0.609 s	0.266 s	13.406 s

Tabulka 6.2: Výsledky testovacího programu `test3_benchmark` na systému MS Windows.

Kapitola 7

Závěr

Práce se zabývala návrhem knihovny a experimentováním s její implementací. Hlavním cílem návrhu bylo jednoduché rozhraní knihovny a tento cíl byl splněn. Přestože možnosti současné verze knihovny nejsou vyčerpávající, počítá návrh knihovny se spoustou rozšíření, které je možné doprogramovat v dalších verzích.

Při implementaci knihovny byly použity moderní programovací techniky, jako jsou šablony jazyka C++ a návrhové vzory. Knihovna funguje (s drobnými odchylkami) stejně na platformách Linux a Microsoft Windows, její rozhraní je pro obě platformy jednotné. Díky testování knihovny na obou platformách byly odhaleny (a opraveny) nejrůznější chyby, které by jinak zůstaly skryty.

Použití knihovny bylo předvedeno na jednoduchých programech, přičemž byl vidět přínos knihovny – práce se sdílenou pamětí se skutečně zjednodušila. Nevýhodou zůstává omezení kladené na typ sdílených objektů. Sdílené objekty nesmí obsahovat virtuální metody, není tedy možné počítat s dynamickým polymorfismem u sdílených objektů.

Sdílená paměť se ukázala jako velmi efektivní forma meziprocesorové komunikace a tato knihovna se snaží programátorům práci se sdílenou pamětí maximálně zjednodušit.

Literatura

[1] *IA-32 Intel® Architecture Software Developer's Manual*. 2004, volume 1: Basic Architecture.

[2] Alexandrescu, A.: *Moderní programování v C++*. Computer Press, 2004, ISBN 80-251-0370-6.

[3] Brown, W. E.: POD Types. http://www.fnal.gov/docs/working-groups/fpcltf/Pkg/ISOcxx/doc/POD.html, 1999.

[4] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1997, ISBN 0-201-63361-2.

[5] Kašpárek, T.; Kočí, R.; Peringer, P.; aj.: Studijní opora k předmětu Operační systémy. http://www.fit.vutbr.cz/study/courses/IOS/public/IOS-opora.pdf, 2006.

[6] Mark Mitchell, A. S., Jeffrey Oldham: *Advanced Linux Programming*. New Riders Publishing, 2001, ISBN 0-7357-1043-0.

[7] Peringer, P.: Seminář C++. https://www.fit.vutbr.cz/study/courses/ICP/public/Prednasky/ICP.pdf, 2004.

[8] Prata, S.: *Mistrovství v C++*. Computer Press, 2004, ISBN 80-251-0098-7.

[9] Ronell, M.: A C++ Pooled, Shared Memory Allocator For The Standard Template Library. http://allocator.sourceforge.net/rtlinux2003.pdf.

[10] Skočovský, L.; Kokolia, V.: *UNIX, POSIX, Plan 9*. první vydání, 1998, ISBN 80-902612-0-5.

[11] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley, special edition vydání, 1997, ISBN 0-201-88954-4.

[12] Sutter, H.; Alexandrescu, A.: *C++ 101 programovacích technik*. Zoner Press, 2005, ISBN 80-86815-28-5.

[13] Wikipedia: Polymorphism in object-oriented programming. http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming, 2007.

[14] Španěl, M.: Webová stránka projektu MDSTk. http://www.fit.vutbr.cz/~spanel/mdstk.

Příloha A

Ukázka práce s knihovnou

Deklarace třídy zapouzdřující vektor bodů, která podporuje sdílení:

```
#include <sharelib.h>
#include <iostream>

// Plain structure
struct Point {
  int x;
  int y;
  Point(int ix, int iy): x(ix), y(iy) { }
};

// Shared object's class
class PointVector: public Share::SharedObject {
public:
  void insert(Point);
  void print();
private:
  // Shared vector container
  typedef Share::Allocator<Point> TAllocator;
  typedef Share::Vector<Point, TAllocator> TVector;
  TVector vector_;
};

// Shareable pointer to class
typedef Share::RelocPtr<PointVector> TPointVectorPtr;
```

Těla metod se nijak neliší od nesdílených objektů:

```
// Insert point p to vector
void PointVector::insert(Point p) {
  vector_.push_back(p);
}

// Print all points to std::cout
void PointVector::print() {
  TVector::const_iterator i;
  for (i=vector_.begin(); i!=vector_.end(); i++) {
    Point p = *i;
    std::cout << ``['' << p.x << ``,'' << p.y << ``]'' << std::endl;
  }
}
```

Konstrukce sdíleného objektu uvnitř nově vytvořeného sdíleného segmentu:

```
const size_t cbSize = 1024 * 1024;
const char szName[] = ``test'';

// Create new shared segment
Share::ShareManager::instance()-> createSegment(szName, cbSize);

// Create new PointVector object inside shared segment
TPointVectorPtr pv = new PointVector;

// Work with shared object
pv-> insert(Point(-1,3));
pv-> insert(Point( 0,2));
pv-> insert(Point( 5,4));

// Get relative object address
TPointVectorPtr::relative_pointer relAddr = pv.toRel();
...
```

Připojení k již existujícímu sdílenému segmentu a práce se sdíleným objektem:

```
const char szName[] = ‘‘test’’;

// Attach existing shared segment
Share::ShareManager::instance()-> attachSegment(szName);

// Create pointer from relative object address
TPointVectorPtr pv = TPointVectorPtr::fromRel(relAddr);

// Work with shared object
pv-> insert(Point(0,0));
pv-> print();
...
```