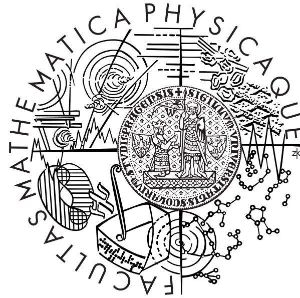
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Dung NGUYEN TIEN

Výuková aplikace zpracování databázových transakcí

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Tomáš Skopal, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika (IOI)

2007

Chtěl bych mnohokrát poděkovat vedoucímu své práce, panu RNDr. Tomáši Skopalovi, Ph.D. za jeho odborné rady a za veškerý čas, který mi s ochotou věnoval.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 28.05.2007. Dung NGUYEN TIEN

Obsah

1 Úvod **7**

2 Implementace **9**

2.1 Platforma 9
2.2 Funkční specifikace 9
2.3 GUI 11
2.4 Test serializovatelnosti 12
2.4.1 Konfliktová serializovatelnost 12
2.4.2 Dvoufázový uzamykací protokol 15
2.4.3 Stromový protokol 18
2.5 Vytváření serializovatelného rozvrhu 20
2.5.1 Dvoufázový uzamykací protokol 20
2.5.2 Wait-Die 23
2.5.3 Stromový protokol 27
2.5.4 Simulace transakčního zpracování 27

3 Uživatelská dokumentace **29**

3.1 Instalace aplikace 29
3.2 Spuštění 29
3.3 File menu 30
3.4 Serializability Test 31
3.4.1 Conflict Serializability Protocol 31
3.4.2 2 Phase Lock Protocol 32
3.4.3 Tree Lock Protocol 32
3.5 Creating Schedule Simulation 33
3.5.1 2 Phase Lock a Wait-Die Protocol 33
3.5.2 Tree Lock Protocol 35
3.6 Pravidla pro ruční zadání 36

3.7 Import souborů 37
3.8 Export souborů 38

4 Závěr **39**
4.1 Budoucnost projektu 40

5 Přílohy **41**
5.1 Instalační CD 41

Literatura **42**

Název práce: *Výuková aplikace zpracování databázových transakcí*
Autor: *Dung NGUYEN TIEN*
Katedra: *Katedra softwarového inženýrství*
Vedoucí bakalářské práce: *RNDr. Tomáš Skopal, Ph.D.*
e-mail vedoucího: *Tomas.Skopal@mff.cuni.cz*

Abstrakt:
Předmětem bakalářské práce je návrh a implementace aplikace pro výuku zpracování databázových transakcí. Aplikace umožňuje simulaci paralelního zpracování transakcí, práci s jednoduchou databází (v paměti). Zpracování transakcí je rozvrhováno buď uživatelem nebo automaticky a je možno jej krokovat (tj. také vidět stavy databáze). Podporuje testování uspořádatelnosti rozvrhu, uzamykací protokoly a protokoly prevence uváznutí.

Klíčová slova: *simulace transakcí, paralelní zpracování, serializovatelnost, rozvrh, databáze*

Title: *Courseware for Database Transaction Processing*
Author: *Dung NGUYEN TIEN*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Tomáš Skopal, Ph.D.*
Supervisor's e-mail address: *Tomas.Skopal@mff.cuni.cz*

Abstract:
The task of the bachelor work is to give a proposal and its implementation of a transaction processing application for teaching purpose. The program enables parallel transaction processing simulation, it works with a simple database in the memory. Transaction processing is scheduled automatically or by user. The processing provides single step operation, this means the user is also able to see actual states of the database. It supports serializability tests, lock protocols and deadlock precaution.

Keywords: *transactions simulation, parallel processing, serializability, schedule, database*

Kapitola 1

Úvod

Pojem transakce již pravděpodobně většina z nás někdy slyšela, mohlo to být v souvislosti s databázemi nebo také z nějaké jiné oblasti, například bankovnictví.

Transakce je jistá posloupnost nebo specifikace posloupnosti akcí, jako jsou čtení, zápis, výpočet, se kterou se zachází jako s jedním celkem [2].

V souvislosti s tímto pojmem se hovoří o tzv. *ACID vlastnostech*, které představují základní princip transakčního zpracování. Jde o první písmena z anglických slov *atomicity*, *consistency*, *isolation* a *durability*:

atomicity - transakce se tváří jako jeden celek, buď se provede celá, nebo vůbec

consistency - transakce převádí databázi z jednoho konzistentního stavu do jiného konzistentního stavu

isolation - nezávislost, transakce jsou nezávislé, dílčí efekty jsou neviditelné jiným transakcím

durability - persistence, úspěšně ukončené transakce jsou uloženy do databáze

Transakce mohou být v programech prováděny paralelně a z hlediska využití zdrojů je přímo žádoucí, aby mohly být zpracovány paralelně různým způsobem. Ovšem libovolné prokládání operací z transakcí vede obecně k

nekonzistentnímu stavu databáze.

Cílem této bakalářské práce je přiblížit fungování mechanismů, které řídí paralelní zpracování transakcí, ať už přes teorii, nebo prostřednictvím simulace transakčního zpracování. V rámci práce vznikla aplikace se jménem **Tsimul** (**T**ransactions **simul**ator). Jeho dalšími úkoly jsou podpora testování serializovatelnosti, implementace uzamykacích protokolů a prevence uváznutí.

Aplikace je určena pro výukové účely v rámci předmětu Databázové systémy přednášeného na MFF UK, a proto jednou ze stěžejních priorit je, aby bylo grafické rozhraní příjemné pro uživatele a ovládání intuitivní. Abychom toho lépe docílili, vybrali jsme vývojové prostředí .NET, kde již základní knihovna obsahuje třídy pro tvorbu grafiky a uživatelského rozhraní.

Pokud jde o principy implementace, v některých případech upřednostňujeme jednoduchost vizualizace na úkor složitosti algoritmů. V následující kapitole (číslo 2) jsou vysvětleny algoritmy testování serializovatelnosti a vytváření rozvrhů. Dále se také podrobněji zabývá GUI, implementací dalších funkcí programu a v závěru modelem simulace transakčního zpracování.

Třetí kapitola obsahuje uživatelskou dokumentaci včetně úplného seznamu dostupných funkcí aplikace. Na konci kapitoly jsou shrnuta pravidla pro ruční zadávání dat do programu, import a export souborů. V závěru je podáno celkové hodnocení a jsou vyjmenována možná vylepšení programu v budoucnosti.

Kapitola 2

Implementace

2.1 Platforma

Program **Tsimul** vyžaduje pro pro svůj běh operační systém *Windows 98* nebo vyšší.

Aplikace je napsána v jazyce *C#*, jde o jazyk vhodný pro vytváření podobných deskových programů v prostředí *Windows*. Spojuje silné vlastnosti jazyka *C* a objektově orientovaného jazyka *Java*. Použitou platformou je prostředí dnes rozšířeného *Microsoft Visual Studio* .NET. V průběhu vývoje byl proveden upgrade aplikace z verze .NET roku 2003 na verzi roku 2005.

2.2 Funkční specifikace

Aplikace používá několik protokolů na testování *uspořádatelnosti* (serializovatelnosti), tedy zda existuje sériový rozvrh ekvivalentní se zadaným. Jednou z možností je test pomocí ***precedenčního grafu***. V tomto případě je otázka serializovatelnosti převedena na řešení úkolu hledání cyklu v acyklickém orientovaném grafu, který je znám pod algoritmem *topologické třídění*. Vše program náležitým způsobem graficky znázorňuje do detailu.

Dále program umožňuje test prostřednictvím ***uzamykacích protokolů***, které jsou založeny na zamykání a odemykání objektů v databázi. V této části byly použity *dvoufázové protokoly*, které v první fázi vše uzamykají a od prvního odemknutí (druhá fáze) se transakcí zamknuté objekty pouze

odemykají. Test je založen na faktu, že jestli jsou transakce *dobře formované* a zároveň *dvoufázové*, pak každý jejich legální rozvrh je serializovatelný.

Jako poslední možnost je testování pomocí ***stromového protokolu***. Tato technika vyžaduje určité znalosti o databázi objektů, které zkoumáme. Jde např. o způsobu fyzického uložení zadané modelem nebo B-stromem, což je náš případ. Na základě *B-stromu* a daného protokolu program zjistí, zda je rozvrh uspořádatelný.

Další moduly ze zadaných transakcí vytváří ***serializovatelné rozvrhy***, pokud existují. První modul používá dvoufázový uzamykací protokol s vlastnostmi uvedenými výše. Je obecně známé, že při jeho užití může nastat tzv. *uváznutí* (deadlock), tj. v jedné transakci chceme uzamknout něco, co již je uzamknuté druhou transakcí. Uváznutí je v aplikaci zobrazeno jako cyklus v *waits-for grafu* (v některých pramenech wait-for graf), který průběžně monitoruje to, zda jednotlivé transakce na sebe nečekají, přesněji řečeno, jestli nečekají na uvolnění objektu uzamčeného jinou transakcí. To je vyznačeno šipkou směřující od čekající transakce k držiteli zámku.

Ukázkou protokolu pro vytváření rozvrhu s ***prevencí uváznutí*** je protokol *Wait-Die*. Ten je založen na dynamickém uzamykání objektů a čekání na jejich uvolnění. V případě, že na objekt čeká transakce s menším tzv. *timestamp* (čas vstupu transakce), je tedy starší, a proto může čekat (wait). Je-li však mladší, je zrušena (abortována) a musí začít znova. Uživatel v této části si může jako v ostatních partiích vše odkrokovat, vidět aktuální stav databáze objektů a zámků.

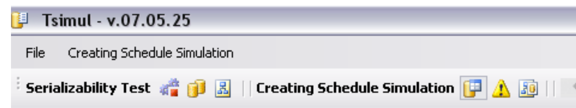
Poslední způsob na vytváření rozvrhu nabízí stromový protokol s využitím již zmíněných principů. Je zde i možnost přímé editace a vytváření hierarchie objektů ve formě B-stromu.

V další sekci této kapitoly nejdříve popíšeme, jak je navrženo grafické uživatelské rozhraní, dále následují sekce zabývající se principy použitých algoritmů při testování uspořádatelnosti a vytváření serializovatelného rozvrhu.

2.3 GUI

Grafické uživatelské rozhraní (GUI) bylo celkově realizováno za účelem prezentace použitých algoritmů v dané oblasti pro účely výuky. Bylo snahou vytvořit prostředí přívětivé a do jisté míry intuitivní pro uživatele, avšak se předpokládá alespoň minimální znalosti z předmětu Databázové systémy učený na nižších stupních studia na vysokých školách.

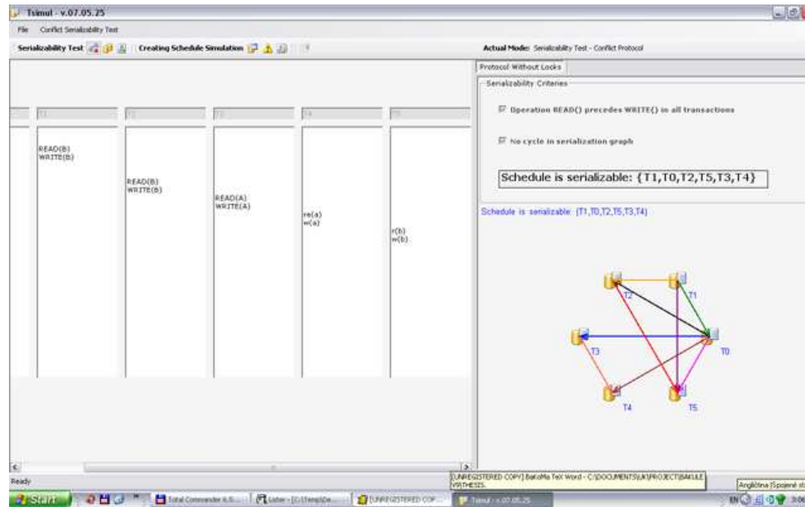
Samotná aplikace obsahuje ***hlavní menu*** pro výběr modu, tedy protokolu, který chceme použít. Každý mod má vlastní menu pro ovládání. Většina často používaných funkcí má své klávesové zkratky zobrazené vedle názvu. Dále se na hlavní ploše nachází *lišta* s nástrojovými tlačítky toolButtons pro rychlé ovládání a přepínání mezi protokoly.



Obrázek 2.1: Menu, nástrojová lišta

Při užití se v pravé části obrazovky zobrazuje ***panel*** s okny zobrazující informace o aktuálním procesu. Při testování serializovatelnosti jsou to vlastnosti daného protokolu, výpis splněných kriterií, výsledek testu a případně i graf serializovatelnosti.

U vytváření serializovatelného rozvrhu se nabízí možnost změny "času" vstupu jednotlivých transakcí do rozvrhu kliknutím na tlačítka. Zásadní je zde možnost si celý průběh vytváření odsimulovat krok po kroku. S tím souvisí i možnost simulace zpět, opět kliknutím tlačítka. Průběžně se při simulaci pro přehled zobrazuje *waits-for graf* se šipkami mezi čekajícím a vlastníkem zámku. Dále je zde zahrnuta funkce *execute*, která rozvrh odsimuluje s průběžným zobrazováním aktuálního stavu databáze proměnných a jejich zámků.



Obrázek 2.2: Aplikace v činnosti

Stromový protokol nabízí navíc možnost graficky intuitivním způsobem vytvořit strom, který je následně aplikován v procesu.

2.4 Test serializovatelnosti

2.4.1 Konfliktová serializovatelnost

První z metod aplikovaných při testování uspořádatelnosti rozvrhů je metoda založena na komutativních vlastnostech operací Read a Write. Teorie komutativity říká, že *operace jsou* ***konfliktní****, jestliže výsledky jejich různého sériového volání nejsou ekvivalentní* [2]. Z této definice pak vyplývá, že následující dvojice operací jsou konfliktní:

> **Read a Write** - tzv. *neopakovatelné čtení*, jde o situaci, kdy jedna transakce zapíše objekt, který byl již před tím přečten jinou transakcí a ta ještě neskončila

Write a Read - tzv. *čtení nepotvrzených dat*, data jsou načtena z jiné transakce, která ještě neskončila

Write a Write - tzv. *přepsání nepotvrzených dat*, transakce přepíše hodnotu, která byla dříve přepsána jinou ještě neukončenou transakcí

V této části předpokládáme, že v každé zadané transakci operace Read předchází operaci Write. Efektivní algoritmy pro případ s volným použitím operace Write neexistují, testování serializovatelnosti je v tomto případě NP-úplný problém.

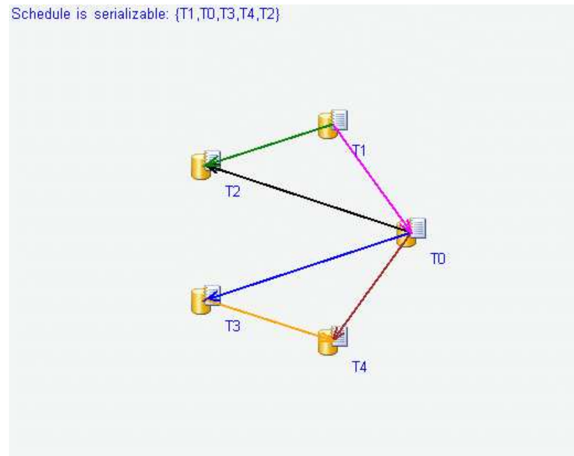
Po načtení vstupního rozvrhu zadaného uživatelem program na základě kompatibility operací sestrojí ***precedenční graf*** rozvrhu, ten je vnitřně reprezentován *maticí sousednosti*. Princip této matice spočívá v tom, že se na horizontální i vertikální osu vynesou postupně všechny vrcholy a v matici se v průsečíku vrcholů zapíše číslo, které určuje počet hran od vrcholu ve vertikální ose k vrcholu na horizontální ose nebo naopak. V grafickém provedení jsou uzly grafu názvy transakcí a mezi nimi vedou hrany *(Ti, Tj)* splněním jedné z podmínek pro určitý atribut:

- transakce *Ti* volá *Read* před tím, než transakce *Tj* volá *Write*
- transakce *Ti* volá *Write* před tím, než transakce *Tj* volá *Read*
- transakce *Ti* volá *Write* před tím, než transakce *Tj* volá *Write*

Dále použijeme následující znalosti:

Tvrzení:
Rozvrh je serializovatelný, jestliže v jeho precedenčním grafu neexistuje cyklus [2].

Tím je problém převeden na hledání cyklu v grafu. Na to aplikujeme algoritmus ***topologického třídění***.



Obrázek 2.3: Precedenční graf

Náš graf uspořádáme následujícím algoritmem:

1. Na začátku máme orientovaný graf G a proměnnou $p = 1$.
2. Najdeme takový vrchol v, ze kterého nevede žádná hrana, říkejme mu *stok*. Pokud v grafu žádný stok není, výpočet končí, protože jsme našli cyklus, tudíž je rozvrh dle uvedeného tvrzení neserializovatelný.
3. Odebereme z grafu vrchol v a všechny hrany, které do něj vedou.
4. Přiřadíme vrcholu v číslo p.
5. Opakujeme kroky 2 až 5, dokud graf obsahuje aspoň jeden vrchol.

Vysvětleme si, proč tento algoritmus funguje. Pokud v grafu nalezneme stok, můžeme mu určitě přiřadit číslo menší než všem ostatním vrcholům, protože překážet by nám v tom mohly pouze hrany vedoucí ze stoku ven a ty neexistují. Jakmile stok očíslujeme, můžeme jej z grafu odstranit a pokračovat číslováním ostatních vrcholů. Tento postup musí někdy skončit, jelikož v grafu je pouze konečně mnoho vrcholů.

Zbývá si uvědomit, že v neprázdném grafu, který neobsahuje cyklus, vždy existuje alespoň jeden stok:

Vezměme libovolný vrchol v_1. Pokud z něj vede nějaká hrana, pokračujeme po ní do nějakého vrcholu v_2, z něj do v_3 atd. Uveďme, co se při tom může stát:

- Dostaneme se do vrcholu v_i, ze kterého nevede žádná hrana. Máme stok.
- Narazíme na v_i, ve kterém jsme už jednou byli. To by ale znamenalo, že graf obsahuje cyklus, což, jak víme, není pravda.
- Budeme objevovat stále nové a nové vrcholy. V konečném grafu to je nemožné.

Celý algorimus má časovou složitost *O(N+M)*, kde N je počet vrcholů a M počet hran.

Jestliže náš graf má topologické uspořádání podle výše uvedeného algorimu, pak neobsahuje cyklus, rozvrh je tedy serializovatelný.

2.4.2 Dvoufázový uzamykací protokol

Vytvářet rozvrhy a následně je testovat s nadějí, že možná budou uspořádatelné může být časově dost neefektivní. Proto byly vynalezeny soubory určitých pravidel (tzv. *protokoly*), podle kterých se sestavují transakce. Za určitých podmínek pak rozvrhy těchto transakcí jsou vždy serializovatelné.

Nejpoužívanější protokoly jsou založené na zamykání a odemykání objektů. Jako první, který jsme aplikovali v našem programu je tzv. ***dvoufázový uzamykací protokol***.

Budeme předpokládat, že atributy v načtených transakcích jsou uzamčené nejvýše jednou transakcí. Pak definujme dle [2]:

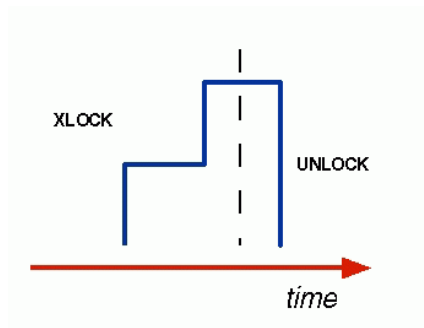
Definice:
Rozvrh je *legální*, jestliže

- atribut je v transakci uzamčený, pokud chce transakce k němu přistupovat
- transakce se nesmí pokoušet zamknout atribut již uzamčený jinou transakcí

Definice:
Transakce je *dobře formovaná*, jestliže

- transakce zamyká atribut, pokud k němu chce přistupovat
- transakce nezamyká atribut, pokud je již touto transakcí uzamčený
- transakce neodemyká atribut, který není touto transakcí uzamčený
- po ukončení transakce, jsou všechny atributy uzamčené v této transakci odemknuté

Definice:
Transakce je *dvoufázová*, jestliže v první fázi uzamyká vše, co je potřeba a od prvního odemknutí (druhá fáze) se transakcí zamknuté atributy pouze odemykají.



Obrázek 2.4: Princip dvoufázovosti

Na základě nahoře uvedených definicí platí:

Tvrzení:
Jestliže všechny transakce jsou dobře formované a dvoufázové, pak každý jejich legální rozvrh je serializovatelný [2].

Této teorie se drží bez výjimky naše aplikace, ověřuje platnost všech pravidel (tj. dvoufázovost, dobře formovanost, legálnost), pokud všechny podmínky platí, rozvrh je serializovatelný.

V první řadě testuje ***dvoufázovost*** transakcí, což není nějak obtížná operace. Pro každou transakci musí platit, že musí mít atributy uzamčené před tím, než nějaký atribut odemkne. Algoritmus projde transakci, jestliže najde operaci odemykání (Unlock), ověří, zda v pozici za ním až do konce se nachází nějaká operace zamykání (X). Pakliže ano, transakce není dvoufázová a test končí s neurčitým výsledkem. V opačném případě pokračuje v testování. Tato operace má v nejhorším případě časovou složitost $O(N^2)$ vzhledem k počtu operací v transakci.

V průběhu testování se ověřuje ***legálnost rozvrhu***. Protokol počítá pouze s exkluzivními zámky už z principu legálnosti, tedy jeden atribut nemůže být ve stejné chvíli použitý více transakcemi.

V další fázi určí, zda jsou transakce ***dobře formované***. To je třeba rozvrh interně odsimulovat. Postupně prochází jednotlivými řádky rozvrhu, určí která transakce je na řadě, vykoná příslušný příkaz. Operace jsou prováděny nad datovou strukturou ve formě pole se dvěma vlastnostmi - název atributu a proměnná pamatující číslo transakce, která drží nad ním zámek.

Pokud selže aspoň jedna z podmínek, transakce nesplňuje náš protokol. V případě kladného výsledku, pro názornost aplikace vykreslí ***precedenční graf***, z něhož je patrné, s jakým sériovým rozvrhem je daný rozvrh ekvivalentní.

Je na místě zdůraznit, že pokud test selže, neznamená to nutně, že daný rozvrh není serializovatelný. Opačná implikace nahoře uvedeného tvrzení neplatí. Lze nalézt legální rozvrh, který není dvoufázový, avšak je uspořádatelný.

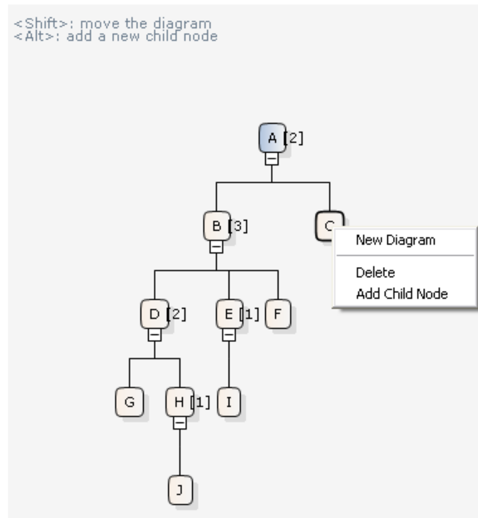
2.4.3 Stromový protokol

Z nedvoufázových protokolů je v této práci zastoupen tzv. ***stromový protokol***. U něj se předpokládá určitá znalost databáze objektů, se kterou se pracuje. Jde např. o způsob přístupu, fyzickou strukturu uložení objektů ve formě hierarchického modelu nebo ve většině případů v B-stromu.

Za předpokladu přítomnosti pouze exkluzivních zámků funguje protokol s těmito pravidly:

- první zámek transakce lze použít na kterémkoli objektu
- další objekt může být uzamčen transakcí, pokud jeho předchůdce byl již uzamčen touto transakcí
- odemykání může probíhat kdykoliv
- objekt, který byl transakcí uzamčen a odemknut, nesmí být opětovně použit stejnou transakcí

Na začátku uživateli nabídne program možnost editace obecného stromu. Uzly jsou atributy nacházející se v databázi. Lze přidávat, odebírat vrchol, nebo měnit celkový vzhled stromu a posouvat ho jako celek. Samozřejmostí jsou i možnosti importu a exportu diagramu.

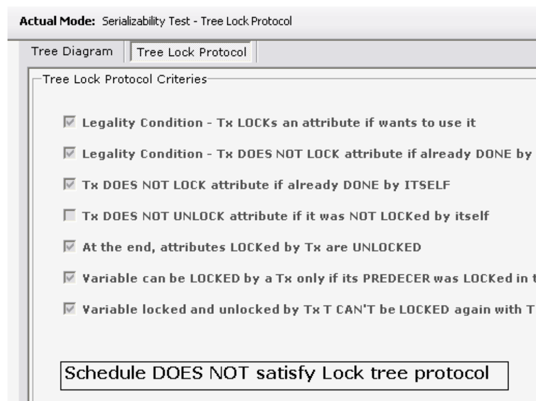


Obrázek 2.5: Návrh stromové struktury

Zde opět nezbývá, než vstupní rozvrh vnitřně odsimulovat a postupně ověřovat platnost podmínek protokolu. Jako hlavní datová struktura je pole atributů nacházejících se v databázi s číslem transakce držící zámek. Dále je třeba si pamatovat, jestli proměnná už byla někdy použijta, tedy uzamknuta a následně odemknuta tou samou transakcí.

Při tomto testu se často používá algoritmus na procházení stromem- *prohledávání do hloubky.* Například při hledání předchůdce aktuálního vrcholu, to šlo samozřejmě obejít rozšířením datové struktury o jednu další vlastnost. Tento algoritmus je všeobecně známý, nebudeme ho zde rozebírat.

Výsledkem testu je kladná odpověď nebo konstatování, že rozvrh nesplnil požadavky protokolu, tudíž nelze o něm rozhodnout, zda je serializovatelný či ne.



Obrázek 2.6: Test serializovatelnosti stromovým protokolem

2.5 Vytváření serializovatelného rozvrhu

2.5.1 Dvoufázový uzamykací protokol

V sekci 2.4.2. jsme použili teorie v ní popsané k testování uspořádatelnosti, zde je aplikujeme na ***vytvoření serializovatelného rozvrhu***.

Potřebujeme 2 základní datové struktury, jedno pole s údaji o proměnných (název, číslo transakce se zámkem) a druhé pole s informacemi o transakcích (číslo transakce, pole atributů které má uzamknuté).

Po načtení transakcí, zadaných ručně nebo importem z textového souboru, program automaticky doplní potřebné zámky. Počítá se s tím, že buď na vstupu nejsou žádné zámky, pak doplní všechny, nebo na vstupu jsou jen některé zámky a zbytek sám dodá. Postupně prochází řádek po řádku aktuální transakce, pokud atribut příkazu ještě nebyl uzamknutý, zamkne jej a na konci všechny objekty odemkne.

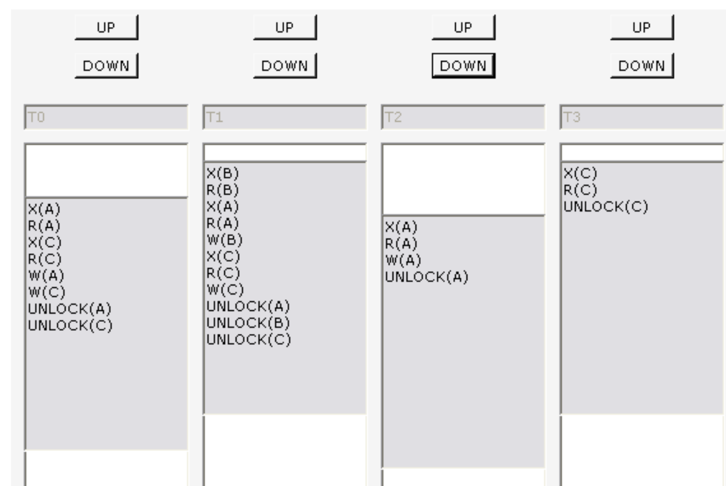
I zde se předpokládají jen exkluzivní zámky (X). Program používá *striktní dvoufázový uzamykací protokol*, oproti "obyčejnému" je "přísnější" v tom, že jsou všechny zámky uvolněny až na konci transakce.

Uživatel má možnost libovolně měnit časy vstupu - tzv. ***timestamp*** jednotlivých transakcí do rozvrhu posouváním celé transakce nahoru a dolů. Ve skutečnosti tedy neměníme časy, ale určujeme, na kterém řádku budoucího rozvrhu vstupují dané transakce do dění. 0. řádek odpovídá abstraktnímu času 0, 1. řádek odpovídá času 1 atd.

Proces vytváření nabídne buď rychlejší variantu, kdy se vytvoří hned ***výsledný rozvrh***, nebo druhou možnost, a to postupnou simulaci od začátku do konce.

Nejdříve se zaplní datová struktura proměnnými ze všech transakcí, pak prochází jednotlivými transakcemi a hledá vhodný příkaz do rozvrhu. Pokud je aktuální čas větší nebo rovno timestampu aktuální transakce a není již na konci, je přidán jeho příkaz do rozvrhu. Příkaz se provede a postupuje se na další řádek.

Jestliže transakce požaduje zámek na volném atributu, dostává ho, v opačném případě je odmítnut a je na řadě další transakce. Je-li to jiný příkaz než zámek, zjistí se v datové struktuře, zda už je proměnná v transakci uzamknuta, pakliže ano, je příkaz vykonán, jinak má přednost jiná transakce. Příkaz odemknutí (Unlock) se provede pouze tehdy, když byl atribut v aktuální transakci uzamknutý.



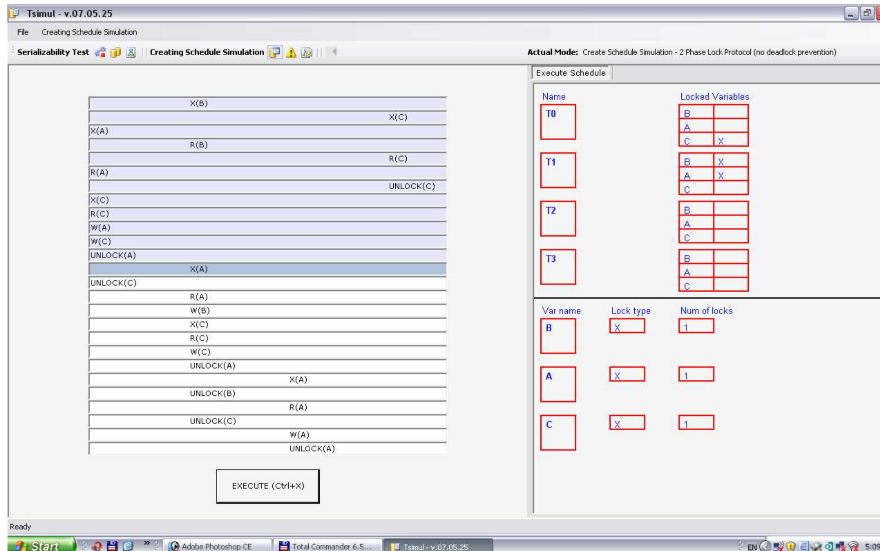
Obrázek 2.7: Posun timestampu

Nejspíš nás napadne otázka, co stane, když ani jedna ze transakcí nemůže vstoupit se svým příkazem do rozvrhu. V tu chvíli nastane ***uváznutí*** (deadlock). Problematika deadlocku není v tomto protokolu zohledňována, řeší ji až protokol v následující kapitole, kterému říkáme *Wait-Die*. V případě, že dojde k uváznutí, algoritmus skončí a ohlásí událost.

Jak ale program pozná, že došlo k deadlocku? Pozná se průběžným udržováním tzv. ***waits-for grafu***, kde uzly jsou aktivní transakce a hrany vedou od čekající transakce k transakci, která drží zámek. Jestliže se v grafu v nějaké chvíli nachází cyklus, došlo k uváznutí a nelze pokračovat ve vytváření rozvrhu. Aplikace se ukončí a vypíše o tom zprávu.

Při zvolení ***simulace "step by step"*** se rozvrh vytváří postupně, krok po kroku, klikáním na tlačítko. Pro názornost je zobrazen i aktuální waits-for graf v pravé části obrazovky.

Pokud bylo vše v pořádku, vytvořený rozvrh lze odsimulovat na proměnných příkazem ***execute***. Napravo se zobrazují tabulky transakcí s proměnnými, které mají právě uzamknuté, a tabulka proměnných s typem zámku, v tomto protokolu výhradně exkluzivní zámek (X). Opět řádek po řádku se mění tyto tabulky a uživatel získá dobrý přehled o aktuálním dění.



Obrázek 2.8: Execute výsledného rozvrhu po krocích se zobrazováním databáze proměnných, zámků a jejich vlastností

2.5.2 Wait-Die

Je na řadě již několikkrát zmíněný protokol ***Wait-Die*** [3]. Jde o jeden z mechanismů pro *prevenci uváznutí*. Většinou se používá v souvislosti se *dvoufázovým umykacím protokolem*, což dělá i naše aplikace. Jde tedy o "upgrade" určitých podmínek předchozího protokolu.

V situaci, kdy o zámek žádá transakce s vyšší prioritou (priorita může být dána různě, záleží na implementaci), v naším případě transakce která je starší(tj. má menší *timestamp*), může ***čekat***. Jestliže má nižší prioritu než transakce držící zámek, pak "umře", taky se říká, že ***abortuje***, a uvolňuje všechny proměnné, začne znova.

Ukažme si, jaké situace mohou nastat:

1. Mějme 3 transakce, každá s jiným timestamp
   T_1: ts = 10

T_2: ts = 20
T_3: ts = 25
T_1 čeká na T_2, T_2 čeká na T_3, T_3 na T_1, pak T_3 "zemře"

2. T_1: ts = 22
   T_2: ts = 20
   T_3: ts = 25

   T_2 čeká na uvolnění atributu A od T_3, přijde T_1 a žádá zámek nad atributem A. Otázkou je, na koho bude čekat T_1. Na T_2 nebo na T_3?

   Jsou dvě možnosti na vyřešení situace:

   - Tou první je, že T_1 čeká na T_3 (transakce, která drží zámek). Ale ve chvíli, když T_2 dostane zámek, T_1 musí "zemřít".

   - Druhá možnost používá následující úvahu:
     T_1 dostane zámek nad A jen po dokončení T_2 a T_3, proto T_1 čeká na obě transakce, pak ale podle pravidel Wait-Die je mladší než T_2 a "umírá".

Teď se podívejme na některé rozdíly oproti přechozímu protokolu bez prevence uváznutí.

Po načtení transakcí program přidá zámky, to je zatím stejné, ovšem může přidat i tzv. ***sdílené zámky*** (S - Share lock).

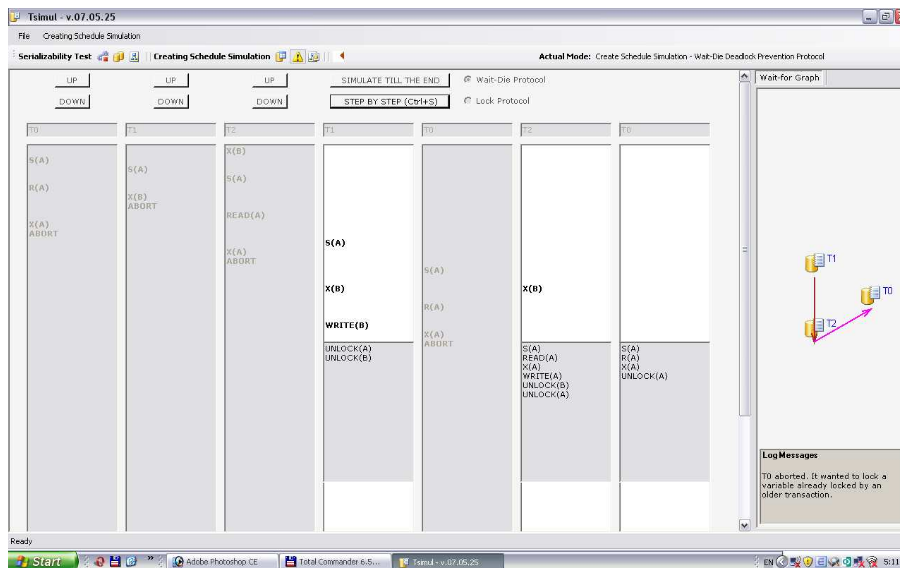
Dvoufázový uzamykací protokol měl jednu, a to velmi podstatnou, nevýhodu. Vždy maximálně jedna transakce v jednu chvíli uzamykala nějaký objekt. Předpokládalo se totiž pouze použítí *exkluzivního zámku.* Wait-Die protokol ale povoluje i *sdílený zámek*, který uzamyká objekty, které chce číst. To ovšem vede k tomu, že jeden atribut může být v jednom okamžiku uzamknutý dvěma nebo více transakcemi tímto zámkem.

Pokud tedy náš algoritmus narazí na příkaz čtení, přidá share zámek, jinak přidá exkluzivní. Jestliže transakce již má sdílený zámek nad atributem a

potřebuje provést zápis, pak musí změnit tento zámek na exkluzivní, jde o tzv. *upgrade zámku.*

Datová struktura se změní přidáním typu zámku, počtu zámků do tabulky proměnných a statutu čekání (zda čeká na uvolnění nějaké proměnné) do tabulky se transakcemi.

Při ***simulaci "krok za krokem"*** je názorně vidět, jak transakce abortují a postupně vzniká nový rozvrh. Opět je zde možné hotový rozvrh odsimulovat a sledovat aktuální držitele zámků a jejich typy.



Obrázek 2.9: Simulace transakčního zpracování po krocích s waits-for grafem

Mezi dalšími *metody prevence uváznutí* patří (více viz.[3]):

Wound-Wait - Zde transakce, která vyžaduje zámek a má vyšší prioritu, "zraní" transakci se zámkem (tj. transakce s nižší prioritou je abortována). Jestliže má nižší prioritu, pak čeká.

Resource ordering - Také známý pod pojmem *konzervativní protokol*. Na začátku žádá zámky všech atributů, které bude potřebovat v celém průběhu. Jestliže nelze získat všechny zámky, aspoň si je "zarezervuje".

Výsledkem této části je ***serializovatelný rozvrh***.

Final Schedule

T0	T1	T2	T2*	T2**	T2***
	S(A)				
	X(B)				
		X(B)			
		ABORT			
S(A)					
	WRITE(B)				
			X(B)		
			ABORT		
R(A)					
	UNLOCK(A)				
				X(B)	
				ABORT	
X(A)					
	UNLOCK(B)				
					X(B)
UNLOCK(A)					
					S(A)
					READ(A)
					X(A)
					WRITE(A)
					UNLOCK(B)
					UNLOCK(A)

Obrázek 2.10: Finální rozvrh simulace

2.5.3 Stromový protokol

Při vytváření rozvrhu *stromovým protokolem* se opět opírá program o principy, se kterými jsme se již seznámili dříve. Všechny legální rozvrhy, u kterých platí pravidla ***stromového uzamykacího protokolu***, jsou serializovatelné.

V podstatě je to opačný postup od zjišťování serializability v tomto protokolu. Postupně ze zadaných transakcí vytváříme rozvrh, který vyhovuje všem pravidlům.

Výsledkem může být rozvrh nebo oznámení, že případ selhal na základě neplatnosti některé z podmínek protokolu. Opět je tu možnost vznik krokově *simulovat* a případně ve finále provést příkaz *execute*.

2.5.4 Simulace transakčního zpracování

V této sekci si vysvětlíme blíže princip fungování ***simulace transakčního zpracování*** při vytváření rozvrhu. (viz. Obrázek 2.9)

Tento proces principiálně je podobný ve všech 3 nahoře zmíněných protokolech. Liší se jen v několika málo detailech. Po té, co uživatel klikne na tlačítko ***SIMULATE TILL THE END***, získá aplikace vnitřně již hotovou podobu finálního zpracování, u *Wait-Die* protokolu to zahrnuje i transakce, které abortovaly. Následná simulace transakčního zpracování je už jen postupné odkrývání udělané práce.

Povězme si, jak to přesně probíhá. V hotovém transakčním zpracování je na každém řádku jeden příkaz. Každý příkaz patří nějaké transakci. Po kliknutí na tlačítko ***STEP BY STEP*** program tímto polotovarem rozvrhu prochází a hledá příkaz, který je na řadě. Zjistí se, které transakci příkaz patří, vymaže se z ní a zapíše se do nově vznikající simulace. Pak se všechny transakce posunou o řádek níže celé, pouze ta naše aktuální transakce jako by "zapomněla" jeden příkaz nahoře.

Pokud právě zapsaným příkazem je ***ABORT***, musí se navíc odsimulovat zrušení transakce. Aktuální pole se zneaktivní (to simuluje akci abort) a aktuální transakce se přemístí do nejbližšího volného místa napravo od ní. Od

této chvíle je přesunutá transakce opět celá se všemi původními příkazy.

Simulace pokračuje, až dokud nebudou všechny transakce prázdné, případně ve dvoufázovém uzamykacím protokolu dojde k uváznutí nebo ve stromovém protokolu dojde k neplatnosti jedné z podmínek.

Kapitola 3

Uživatelská dokumentace

3.1 Instalace aplikace

Přibalené instalační CD obsahuje 2 balíky, jeden pro spuštění aplikace, druhý se všemi zdrojovými kódy programu v jazyce *C#* napsaném na platformě *Microsoft Visual Studio* .NET 2005 pro prostředí *Windows 98* a výše.

Aplikace je zabalena v instalačním balíku *TsimulSetup.zip*. Po rozbalení najdeme adresář *Documentation* se specifikací projektu, uživatelskou dokumentaci v českém jazyce a zjednodušenou verzi v anglickém jazyce. Dále adresář *Icons* s programovými ikony, adresář *Testing Datas*, který obsahuje příklady pro testování funkčnosti programu, konkrétně jde o rozvrhy, transakce a stromové diagramy všech typů. V hlavním adresáři se také nachází exe soubor ke spuštění aplikace - *Tsimul.exe*, knihovna *Lithium.dll* a aplikační ikona *App.ico*.

3.2 Spuštění

Pro spuštění programu se použije soubor *Tsimul.exe*. Po spuštění se zobrazí hlavní okno typu deskové aplikace. V horní části obrazovky je menu a pod ním je nástrojová lišta pro přepínání mezi protokoly. Nahoře v pravém rohu je label upozornění, který modul se bude používat.

3.3 File menu

Kliknutím na ***File menu*** se nabídnou tyto možnosti:

- *Select Mode* (Ctrl+M)
  Po výběru se zobrazí dialogové okno, na kterém uživatel vybere příslušný protokol (mode):

  - *Serializability Test*
    Pro testování serializovatelnosti rozvrhu

    Test obsahuje tyto podmožnosti:

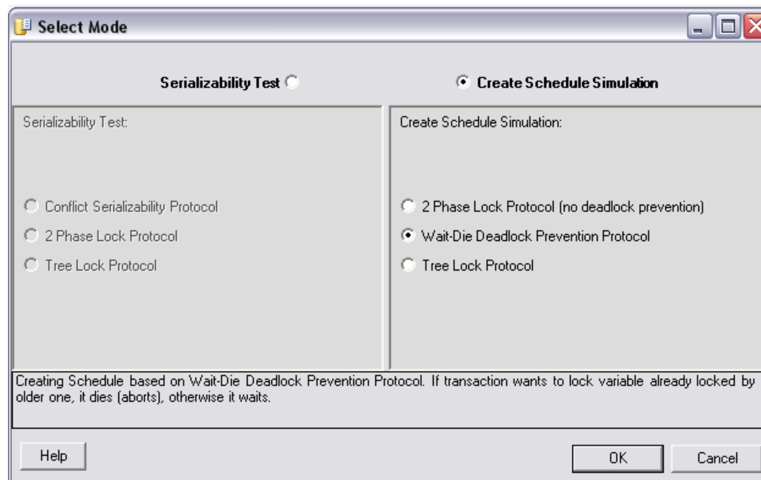
    * *Conflict Serializability Protocol* - Testování pomocí konfliktové serializovatelnosti
    * *2 Phase Lock Protocol* - Testování pomocí dvoufázového uzamykacího protokolu
    * *Tree Lock Protocol* - Testování pomocí stromového protokolu

  - *Creating Schedule Simulation*
    Simulace vytváření serializovatelného rozvrhu

    Simulace obsahuje tyto podmožnosti:

    * *2 Phase Lock Protocol (no deadlock prevention)* - Simulace vytváření pomocí dvoufázového uzamykacího protokolu
    * *Wait-Die Deadlock Prevention Protocol* - Simulace vytváření pomocí Wait-Die protokolu s prevencí uváznutí
    * *Tree Lock Protocol* - Simulace vytváření pomocí stromového protokolu

  Tlačítko *OK* potvrdí volbu, *Cancel* zruší volbu.



Obrázek 3.1: Výběr protokolu

- *Exit* (Atl+F4)
  Pro ukončení programu. Po výběru se zobrazí ujišťovací dotaz, zda opravdu ukončit.

3.4 Serializability Test

3.4.1 Conflict Serializability Protocol

Po vybrání tohoto protokolu se zobrazí ***Conflict Serializability Test menu*** a to nabízí tyto možnosti:

- *New Test* - Vyprázdní pracovní plochu a připraví aplikaci k novému testu
- *Add Transaction* (Ctrl+T) - Přidání okna pro další transakci (ruční zadání). Pro podrobnosti o pravidlech ručního zadání viz. 3.6.
- *Import Schedule* (Ctrl+I) - Import rozvrhu z textového souboru. Pro podrobnosti o pravidlech struktury textového souboru s rozvrhem viz. 3.7.

- *Read Schedule* (Ctrl+R) - Načtení dat v textovém poli do paměti
- *Serializability Test* (Ctrl+S) - Test serializovatelnosti rozvrhu
- *Export Serialization Graph* (Ctrl+E) - Export precedenčního grafu

3.4.2 2 Phase Lock Protocol

Vybráním tohoto protokolu se zobrazí ***2PL Serializability Test menu***, které nabízí tyto možnosti:

- *New Test* - Vyprázdní pracovní plochu a připraví aplikaci k novému testu
- *Add Transaction* (Ctrl+T) - Přidání okna pro další transakci (ruční zadání). Pro podrobnosti o pravidlech ručního zadání viz. 3.6.
- *Import Schedule* (Ctrl+I) - Import rozvrhu z textového souboru. Pro podrobnosti o pravidlech struktury textového souboru s rozvrhem viz. 3.7.
- *Read Schedule* (Ctrl+R) - Načtení dat v textovém poli do paměti
- *Well-formed Control* (Ctrl+W) - Kontrola, zda jsou transakce v rozvrhu dobře formované
- *Two-phased Control* (Ctrl+P) - Kontrola, zda jsou transakce v rozvrhu dvoufázové
- *Serializability Test* (Ctrl+S) - Test serializovatelnosti rozvrhu
- *Export Serialization Graph* (Ctrl+E) - Export precedenčního grafu

3.4.3 Tree Lock Protocol

Vybráním tohoto protokolu se zobrazí:

- ***Edit Tree Diagram menu***
  - *New* (Ctrl+Alt+D) - Nový diagram

- *Random Diagram* (Ctrl+Alt+R) - Náhodný generátor vytvoří nový diagram
- *Add Child Node* (Ctrl+Alt+A) - Přidání potomka k aktivnímu uzlu
- *Delete Node* - Mazání uzlu
- *Import Diagram* - blíže viz. 3.7.
- *Export Diagram* - blíže viz. 3.8.
- *Close* - Zavře diagram

V pravé části obrazovky se zobrazuje okno *Diagram Properties*, kterým lze měnit vlastnosti diagramu. Diagram se může modifikovat i kliknutím na uzel pravým tlačítkem myši, zobrazí se malé menu s funkcemi: *New Diagram*, *Delete* a *Add Child Node*. Jejich význam je uveden výše.

- ***Tree Lock Serializability Test menu***
  - *New Test* - Nový test
  - *Add Transaction* (Ctrl+T) - Přidání okna pro další transakci (ruční zadání). Pro podrobnosti o pravidlech ručního zadání viz. 3.6.
  - *Import Schedule* (Ctrl+I) - Import rozvrhu z textového souboru. Pro podrobnosti o pravidlech struktury textového souboru s rozvrhem viz. 3.7.
  - *Read Schedule* (Ctrl+R) - Načtení dat v textovém poli do paměti
  - *Serializability Test* (Ctrl+S) - Test serializovatelnosti rozvrhu

3.5 Creating Schedule Simulation

3.5.1 2 Phase Lock a Wait-Die Protocol

Po vybrání protokolu *2 Phase Lock Protocol* nebo *Wait-Die Deadlock Prevention Protocol* se objeví

- ***Creating Schedule Simulation menu***:
  - *New Simulation* - Nová simulace vytváření rozvrhu
  - *Add Transaction* (Ctrl+T) - Přidání okna pro další transakci (ruční zadání). Pro podrobnosti o pravidlech ručního zadání viz. 3.6.
  - *Import Transactions* (Ctrl+I) - Import transakcí z textového souboru. Pro podrobnosti o pravidlech struktury textového souboru s rozvrhem viz. 3.7.
  - *Read Transactions* (Ctrl+R) - Načtení dat v textovém poli do paměti

    Po načtení lze zvolit protokol, který chce uživatel použít, a to kliknutím na jeden z radioButtonů: *Wait-Die Protocol* nebo *Lock Protocol* (Dvoufázový uzamykací protokol)

  - *Create Serializable Schedule* (Ctrl+C) - Vytvoření serializovatelného rozvrhu

    Po vybrání možnosti *Create Serializable Schedule* program přidá příslušné zámky do transakcí.

    Uživatel může posouvat začátky vstupů transakcí do rozvrhu použitím tlačítek *UP* pro pohyb transakce nahoru a *DOWN* dolů.

    Pak lze zvolit jedno z tlačítek:

    * *SIMULATE TILL THE END* - simulace vytváření rozvrhu až do konce, zobrazí se finální serializovatelný rozvrh

    * *STEP BY STEP* (Ctrl+S) - simulace krok po kroku, při této činnosti se v pravé části obrazovky průběžně zobrazuje a aktualizuje waits-for graf. Dále v nástrojové liště se zaktivuje tlačítko *Back* pro zpětnou simulaci

    * *Export Schedule* (Ctrl+E) - Export rozvrhu

        * *Execute Schedule* (Ctrl+X) - Execute rozvrhu, v pravé části obrazovky jsou po každém kliknutí executu aktualizovány tabulky uzamknutých atributů, typů zámků, počtů zámků nad proměnnými

3.5.2 Tree Lock Protocol

- ***Edit Tree Diagram menu:***
    - Viz. 3.4.3.
- ***Creating Schedule Simulation menu:***
    - *New Simulation* - Nová simulace vytváření rozvrhu
    - *Add Transaction* (Ctrl+T) - Přidání okna pro další transakci (ruční zadání). Pro podrobnosti o pravidlech ručního zadání viz. 3.6.
    - *Import Transactions* (Ctrl+I) - Import transakcí z textového souboru. Pro podrobnosti o pravidlech struktury textového souboru s rozvrhem viz. 3.7.
    - *Read Transactions* (Ctrl+R) - Načtení dat v textovém poli do paměti
    - *Create Serializable Schedule* (Ctrl+C) - Vytvoření serializovatelného rozvrhu

      Po vybrání možnosti *Create Serializable Schedule* program přidá příslušné zámky do transakcí, pokud ještě nebylo provedeno předtím.
      Uživatel může posouvat začátky vstupů transakcí do rozvrhu použitím tlačítek *UP* pro pohyb transakce nahoru a *DOWN* dolů.

      Pak lze zvolit tlačítko:

        * *SIMULATE TILL THE END* - simulace vytváření rozvrhu až do konce, zobrazí se finální serializovatelný rozvrh

Potom se zobrazí tlačítko:

· *STEP BY STEP* (Ctrl+S) - simulace krok po kroku, při této činnosti se v pravé části obrazovky průběžně zobrazuje a aktualizuje waits-for graf. Dále v nástrojové liště se zaktivuje tlačítko *Back* pro zpětnou simulaci.

· *Export Schedule* (Ctrl+E) - Export rozvrhu

· *Execute Schedule* (Ctrl+X) - Execute rozvrhu, v pravé části obrazovky jsou po každém kliknutí executu aktualizovány tabulky uzamknutých atributů, typů zámků a počtů zámků nad proměnnými

3.6 Pravidla pro ruční zadání

- Přípustné příkazy:
  - *Read(proměnná)* - načtení proměnné
  - *Write(proměnná)* - zápis proměnné
  - *X(proměnná)* - exkluzivní (výlučný) zámek nad proměnnou
  - *S(proměnná)* - Share lock (sdílený zámek) nad proměnnou
  - *Unlock(proměnná)* - odemknutí proměnné

  Při zadání stačí psát počáteční písmena příkazů.

  Jakékoliv další příkazy nad určitou proměnnou jsou považovány za příkazy, které jsou typu read-only, neprovádějí zápis do proměnné.

  Načítání je case-insensitive, nerozlišuje tedy mezi malými a velkými písmeny. Proměnné pro zjednodušení načítání musí mít právě jeden znak.

- Pravidla pro zadávání rozvrhů:
  - Minimální počet transakcí v jednom rozvrhu je 2.
  - Každý řádek rozvrhu obsahuje právě jeden příkaz.
  - Rozvrh v *Conflict Serializability Test* nesmí obsahovat zámky, tj. nesmí mít žádný z příkazů: *X(proměnná)*, *S(proměnná)*, *Unlock(proměnná)*.
  - Rozvrh v *2PL Serializability Test* nesmí mít Share lock.
  - Rozvrh ve *stromovém protokolu* nesmí mít Share lock

- Pravidla pro zadávání transakcí:
  - Minimální počet transakcí v jednom rozvrhu je 2.
  - Každý řádek každé transakce obsahuje právě jeden příkaz.
  - Transakce v *dvoufázovém uzamykacím protokolu* nesmí obsahovat Share lock.
  - Transakce ve *stromovém protokolu* nesmí obsahovat Share lock.

V případě porušení některé z "mírnějších" pravidel program oznámí, vyspecifikuje chybu a umožňuje nápravu. Došlo-li k závažnějšímu porušení (tyto chyby vzniknou většinou v důsledku nekorektního jednání uživatele), nahlásí událost a ukončí protokol.

3.7 Import souborů

- Pravidla pro importované soubory transakcí:
  - Soubory musí být formátu *txt* souborů. Jiné typy se v nabídce importu nezobrazí.
  - Všechny transakce jsou zapsané do jednoho sloupce.
  - Transakce jsou od sebe oddělené právě jedním prázdným řádkem.
  - Na každém řádku souboru je právě jeden příkaz.

- Pravidla pro importované soubory rozvrhů:
  - Soubory musí být formátu *txt* souborů. Jiné typy se v nabídce importu nezobrazí.
  - Na každém řádku je právě jeden příkaz.
  - Sloupce v rozvrhu jsou odsazené tabulátory nebo mezerami, příkazy jednoho sloupce musí mít stejné odsazení.

Importované diagramy ve *stromovém protokolu* jsou typu *gxl*. Lze je navrhnout ve stromovém protokolu, podle bodu 3.4.3.

3.8 Export souborů

Precedenční graf rozvrhu při testování je exportován ve formátu *jpg* souboru.

Serializovatelný rozvrh vytvořený z načtených transakcí je exportován ve formátu *txt* souboru.

Diagram vytvořený ve *stromovém protokolu* je exportován ve formátu *gxl* souboru.

Kapitola 4

Závěr

Cílem práce bylo vytvořit aplikaci pro účely výuky oblasti paralelního zpracování transakcí. Aplikace měla umožnit simulaci vytváření serializovatelných rozvrhů, jeho krokování a podporovat testování uspořádatelnosti. Měli být zavedeny uzamykací protokoly a prevence uváznutí. Stavy databáze měly být viditelné uživateli.

Všechny body ze specifikace projektu se podařilo naplnit. I když některé části mohly být ještě lépe realizovány. Naproti tomu však aplikace během vývoje se rozrostla významným způsobem, a to hlavně díky odbornému vedení vedoucího projektu. Oproti původním předpokladům přibyl kompletní stromový protokol, vykreslování waits-for grafu, možnost spuštění rozvrhu příkazem execute, atd.

Práce na projektu byla časově velmi náročná, zabrala mi opravdu hodně času. Přesto všechno jsem se ale naučil mnoha věcí, které bych se nikdy nenaučil, dostal se k zajímavým pramenům teorie o transakcích a databázích obecně. Jsem proto velmi rád, že jsem byl součástí tohoto projektu.

4.1 Budoucnost projektu

V budoucnu by program **Tsimul** mohl být rozšířen o některé z dalších funkcí:

- Spojováním konfliktních dvojic operací šipkami přímo v rozvrhu pro větší názornost
- Vyznačení *deadlocku* do rozvrhu
- Možnost posouvání příkazů v transakcích táhnutím myši
- Vyznačení timestampu
- Použití tlačítka *Back* i při *executu*
- Přidání dalších zajímavých protokolů (*Wound-Wait*, *Časová razítka*,. . .)

Kapitola 5

Přílohy

5.1 Instalační CD

Přílohou je přiložené CD, na kterém jsou:

- Zdrojové kódy celé aplikace v jazyce *C#* pro platformu *Microsoft Visual Studio* .NET 2005
- Instalační balík

Literatura

[1] Haritsa J., Carey M., Livny M.: *Data access scheduling in firm real-time database systems*, Kluwer Academic Publishers, Netherlands, 1992.

[2] Pokorný J., Žemlička M.: *Základy implementace souborů a databází*, UK Praha, Karolinum, 2004.

[3] Rosenkrantz D., Stearns R., Lewis P.: *System level concurrency control for distributed database systems*, ACM Press, USA, 1978.

[4] Pokorný J., Halaška I.: *Databázové systémy*, FEL ČVUT, Praha, 2003.

[5] Liberty J.: *Programming C#: Building .NET Applications with C#*, O'Reilly Media, Inc., 2005.

[6] *C# Station Home Page, http://www.csharp-station.com/.*

[7] Dekhtyar A.: *Home Page, http://www.cs.uky.edu/˜dekhtyar/.*

[8] Skopal T.: *Home Page, http://urtax.ms.mff.cuni.cz/˜skopal/.*

[9] *Distributed Systems Research Group, Charles University, Prague, http://nenya.ms.mff.cuni.cz/˜ceres/txy/.*