VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

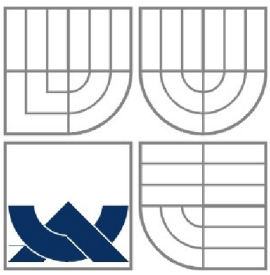
GARBAGE COLLECTOR OBJEKTŮ JAZYKA PNTALK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

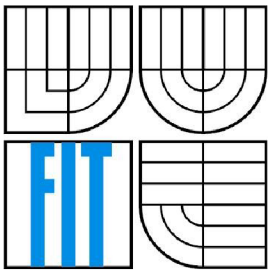
AUTOR PRÁCE
AUTHOR

Bc. JÁN MIŠÁK

BRNO 2016



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

GARBAGE COLLECTOR OBJEKTŮ JAZYKA PNTALK

GARBAGE COLLECTOR FOR PNTALK OBJECTS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JÁN MIŠÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK KOČÍ, Ph.D.

BRNO 2016

Abstrakt

Tato práce se zabývá návrhem a implementací garbage collectoru pro virtuální stroj jazyka PNtalk. Jsou v ní popsány a zhodnoceny jednotlivé přístupy a konkrétní algoritmy pro automatickou správu paměti a detailne popsáno jejich využití v praktické implementaci pro vitruálne stroje PNtalku, které jsou implementovány v Smalltalku a C++. Práce uvádí čtyři rodiny algoritmů, a to rodiny mark-sweep, mark-compact, kopírovací algoritmy a počítání referencí. Nejprve jsou popsány sekvenční verze algoritmů, které zastavují běh hlavního programu (mutátoru), pak jsou uvedeny jejich paralelní a nakonec konkurentní verze, které běh mutátoru nezastavují. Práce též uvádí generační model garbage collectingu.Výsledkem práce je vypracování generačního garbage collectoru pro jazyk PNtalk, vypracování testů a na jejich zakladě odměření optimálních parametrů.

Abstract

This thesis deals with the designing of a garbage collector for the PNtalk virtual machine. It describes and rates the approaches and algorithms for an automatic memory management. Four algorithm families ale presented: mark-sweep, mark-compact, copying algorithms and reference counting. At first it describes sequential forms, that pauses running of the main program (mutator), then it describes parallel and concurent forms, that do not pauses the mutator. The thesis also presents generational model of garbage collecting. The following sections briefly introduces object orientated Petri nets. The result of this thesis is the design of the generational garbage collector for the PNtalk virtual machine.

Klíčová slova

PNtalk, garbage collector, zpráva paměti, mark-sweep, mark-compact, kopírovací garbage collector, počítání referencí, generační garbage collector, Petriho síť

Keywords

PNtalk, garbage collector, memory management, mark-sweep, mark-compact, copying garbage collection, reference counting, generational garbage collection, Petri net

Citace

Ján Mišák: Garbage collector objektů jazyka PNtalk, diplomová práce, Brno, FIT VUT v Brně, 2016

Garbage collector objektů jazyka PNtalk

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Radka Kočího, Ph.D.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Ján Mišák
10.1.2016

Poděkování

Chcel by som sa poďakovať svojmu vedúcemu Ing. Radkovi Kočímu, Ph.D. za množstvo odborných rád, podnetov, nápadov a za pomoc, ktorú mi pri tvorbe práce poskytol. Taktiež mu chcem poďakovať za poskytnutie študijných materiálov potrebných pre vypracovanie práce.

Obsah

1 Úvod

Dnešnou snahou vo vývoji programovacích jazykov je poskytnúť programátorom nástroje pre zefektívnenie ich práce. To sa dá docieliť zlepšením čitateľnosti zápisu a zmenšením objemu potrebného kódu. Moderné programovacie jazyky sa to snažia dosiahnúť čo najväčšou abstrakciu od potrieb architektúr hardvéru, a tým chcú docieliť možnosť čistejčieho zápisu algoritmov a menšiu chybovosť kódu. Dobrým príkladom tejto snahy je vznik algoritmov pre automatickú správu pamäti, ktorú dnes implementuje veľká časť programovacích jazykov.

Druhou oblasťou vývoja programovacích jazykov je zvyšovanie ich modelovacej schopnosti. Dnešné hardvérové architektúry ponúkajú stále lepšiu podporu pre beh paralelných algoritmov. Vzniká teda prirodzená potreba pre nástroje umožňujúce efektívny vývoj a simulovanie paralelných systémov. Existuje niekoľko matematických formalizmov na popis paralelných procesov. Jedným z nich sú Petriho siete, ktoré predstavujú vhodný nástroj pre modelovanie diskrétnych paralelných systémov. Petriho siete spájajú výhody zrozumiteľného grafického zápisu a možnosti simulácie s dobrou formálnou analyzovateľnosťou. Práve na tomto formalizme bol založený jazyk PNtalk [4].

Dnes sa na FIT VUT v Brne vyvýjajú dve implementácie simulátoru. Jedna je v C++ a druhá v jazyku Smalltalk. Táto práca je súčasťou tohto vývoja a rozširuje ho o automatickú správu pamäte, ktorá zatiaľ nebola v simulátoroch implementovaná.

1.1 Ciele a štruktúra práce

Práca má dva ciele. Navrhnúť, implementovať a otestovať GC pre existujúce implementácie jazyka PNtalk a to implementáciu v C++ a v jazyku Smalltalk. Druhý cieľ je vytvoriť sadu testov pre overenie správnosti fungovania garbage collectoru (ďalej len GC) a pre nájdenie čo najlepších parametrov pre použité algoritmy. Navrhnutý GC musí vyhovovať vlastnostiam a zameraniu daného jazyka tak, aby bol vyvážený potrebný výkon a pamäťové požiadavky pre programy napísané v PNtalk. Keďže jazyk Smalltalk obsahuje jeho vlastný GC, v tomto prípade pôjde len o efektívne odstránenie všetkých referencií na dané objekty z vnútorných štruktúr PNtalku. Pri C++ implementácii tento krok bude ďalej rozšírený o fyzické alokovanie a uvoľňovanie pamäte.

Práca je rozdelená do deviatich kapitol, pričom prvá kapitola je úvodná. V druhej kapitole je popísaná problematika správy pamäte. Nasledujú dve kapitoly, ktoré sa zaoberajú jednotlivými prístupami v automatickej správe pamäte. Piata kapitola predstavuje jazyk PNtalk. V šiestej kapitole je predstavený návrh *garbage collectoru* pre jazyk PNtalk. Siedma a ôsma kapitola popisuje implementáciu a testovanie navrhnutého GC a v záverčnej deviatej kapitole je uvedené zhodnotenie práce.

2 Automatická správa pamäte

Jednou z vlastností moderných programovacích jazykov, ktorá dokáže značne urýchliť vývoj programov a znížiť ich chybovosť, je automatická správa pamäte. Táto kapitola predstavuje úvod do tejto problemaky. Zaoberá sa príčinami potreby automatickej správy a predstavuje najčastejšie metriky pre hodnotenie kvality riešení. V tejto kapitole som čerpal zo zdrojov [1] a [2].

2.1 Dynamická alokácia pamäte

Takmer všetky moderné jazyky používajú dynamickú alokáciu pamäte. To umožňuje, aby objekty boli dynamicky vytvárané a rušené aj napriek tomu, že vopred, v čase prekladu, nepoznáme ich veľkosť. Dynamicky alokovaný objekt je najčastejšie uložený na halde. Alokácie na halde sú veľmi dôležité, pretože poskytujú možnosti ako:

- Dynamicky stanoviť veľkosť nových objektov, čo umožňuje predísť pádom programov kvôli prekročeniu hraníc staticky alokovaného pamäťového priestoru.
- Definovať rekurzívne dátové štruktúry ako sú zoznamy, stromy a mapy.
- Vrátiť novo vytvorené objekty rodičovskej procedúre (napríklad návrhový vzor továreň)
- Vrátiť funkciu ako výsledok inej funkcie (napríklad *closure* alebo *suspension* vo funkcionálnych jazykoch)

Objekty alokované na halde sú prístupné cez referencie. Referencia je primárne chápaná ako ukazovateľ na daný objekt (jeho počiatočná adresa v pamäti). Avšak, referencia môže ukazovať na objekt aj nepriamo. Napríklad cez *handle*, ktorý následne ukazuje na daný objekt. *Handle* umožňujú realokovať objekt (aktualizovaním *handlu*) bez potreby meniť všetky ukazovatele v programe na daný objekt. Akýkoľvek program bežiaci na architektúre s konečným množstvom pamäte musí občas obnoviť alebo znovu použiť pamäť, ktorá je zabraná objektami, ktoré už nie sú potrebné. Pamäť na halde môže byť vrátená operačnému systému buď explicitne (v jazyku C volaním funkcie *free*) alebo automaticky, kde podpora tejto možnosti musí byť implementovaná v behovom prostredí daného programu.

2.2 Explicitné uvoľňovanie pamäte

Pri tomto prístupe má celú prácu s alokovaním a uvoľňovaním pamäte vo svojich rukách programátor. To môže viesť k zavedeniu dvoch druhov chýb do programov. Programátor môže uvoľňovať pamäť predčasne, to je ak v programe existuje referencia na uvoľňovaný objekt, cez ktorú bude v ďalšej časti programu k objektu ešte pristupované. Výsledok operácie prístupu cez takúto referenciu je nepredikovateľný, pretože programátor väčšinou nemá priamu kontrolu nad tým, čo sa stane s daným kusom pamäte po volaní funkcií pre jeho uvoľnenie. Behové prostredie môže buď uvoľnenú pamäť vynulovať alebo ju okamžite použiť pre alokovanie nového objektu, prípadne ju hneď vrátiť operačnému systému. Druhý typ chyby je, že programátor neuvoľní použitú a ďalej nepotrebnú pamäť, čo vedie k úniku pamäte. V krátko bežiacich aplikáciách to nemusí byť problém, no v dlhodobo bežiacich alebo pamäťovo náročných aplikáciách môže spôsobovať spomalenia až nestabilitu aplikácií.

Tieto chyby sa často vyskytujú, ak existuje referencia na daný objekt v dvoch a viac rôznych objektoch. Tento problém je ešte viac citeľný pri programoch, ktoré využívajú viacero vláken, ktoré si držia referenciu na daný objekt. Problém nesprávnej správy pamäte nie je len vecou nepozornosti programátora. Často to je samostatný programátorský problém.

2.3 Automatické uvoľňovanie pamäte – garbage collection

Vyššie uvedené problémy sú riešiteľné použitím behového prostredia s automatickou správou pamäte nazývanou pojmom *garbage collection*. Táto vlastnosť behového prostredia zabraňuje predčasnému uvoľneniu pamäte. Objekt je uvoľnený iba ak neexistuje referencia, ktorá by k nemu dokázala pristúpiť. Takýto objekt je nazývaný nedosiahnuteľný objekt. Rovnako garantuje uvoľnenie všetkých nepotrebných objektov. Každý nedosiahnuteľný objekt bude skôr alebo neskôr zozbieraný a uvoľnený *garbage collectorom* (GC). Jeho úlohou je identifikovať nedosiahnuteľné časti alokovanej pamäte, ktoré sú nazvané *garbage*, alokovať miesto pre nové objekty, zozbierať nedosiahnuteľné objekty a uvoľniť miesto nimi zabrané. Program bežiaci v behovom prostredí, pre ktorý GC spravuje pamäť sa nazýva mutátor. V ďalších kapitolách uvažujeme aj konečnú množinu koreňových ukazovateľov mutátora, ktoré sú prístupné priamo z mutátora teda bez potreby prístupu cez iné objekty. Objekty, ktoré sú prístupné cez koreňové ukazovatele nazývame koreňové objekty mutátora (*roots*). Je dôležité spomenúť pojem živé objekty. Sú to objekty, ku ktorým v budúcnosti mutátor ešte pristúpi. Problém živosti objektov je čiastočne rozhodnuteľný. Dá sa aproximovať vlastnosťou nazvanou dosiahnuteľnosť. Objekt je dosiahnuteľný, ak existuje cesta od koreňových ukazovateľov až po daný objekt. Je však potrebné pamätať na dva faktory. Prvý je, že aproximácii pomocou dosiahnuteľnosti nie vždy vyhovujú všetky objekty, ku ktorým už nebude pristúpené. Druhým faktorom je výkonnosť. Nie vždy je výhodné uvoľniť všetky nedosiahnuteľné objekty a preto niektoré nedosiahnuteľné objekty ostávajú v pamäti aj po tom, ako sa stali nedosiahnuteľnými.

Keďže o správu sa stará behové prostredie, odpadá aj problém pokusu o viacnásobné uvoľnenie pamäte. Celá problematika správy pamäte je teda presunutá na GC, ktorý si uchováva globálnu štruktúru alokovaných objektov na halde a referencie, cez ktoré k nim môže pristupovať. Ďalšou výhodou automatickej správy pamäte je, že minimalizuje rozhrania medzi jednotlivými časťami programov. Jednotlivé moduly programu sa nemusia starať o objekty, ktoré môžu byť využívané inými modulmi. Tým narastá zrozumiteľnosť a čitateľnosť zdrojových kódov a zvyšuje sa možnosť miery zapuzdrenia.

Problémom automatickej správy pamäte sú zvýšené nároky na výpočtové zdroje, ktoré sa týkajú hlavne procesorového času, no niekedy aj vyššieho nároku na spotrebovanú pamäť. Na druhej strane sú časté prípady, kedy zavedenie *garbage collectoru* zrýchli beh programu. Týmito prípadmi sa zaoberá 3. kapitola.

2.4 Lokalita behu programu

Každý výpadok stránky alebo cache zapríčiňuje časovú penalizáciu v behu programu. Ak má program dobrú lokalitu, potom počet týchto výpadkov je nízky. Rozlišujeme dva druhy lokality:

- **Časová lokalita** – zachytáva lokalitu dát v čase. Ak má program dobrú časovú lokalitu, tak ak bolo pristúpené k nejakým dátam, tak je vysoká pravdepodobnosť, že k nim bude pristúpené v blízkej budúcnosti opäť.
- **Priestorová lokalita** – zachytáva lokalitu dát v priestore. Ak bolo pristúpené k istým dátam, tak je vysoká pravdepodobnosť, že sa v blízkej budúcnosti pristúpi k dátam, ktoré sú fyzicky uložené na blízkych adresách.

Ak má program dobrú časovú lokalitu, tak v najlepšom prípade stránky obsahujúce všetky dáta, ku ktorým sa pristupuje, sa nachádzajú v hlavnej pamäti, a teda neprichádza ku výpadkom stránok. Ak aj náhodou dôjde k výpadku stránky, algoritmy pre výmenu stránok v pamäti ako LRU (last recently used) dokážu v takom prípade pracovať veľmi efektívne. Priestorová lokalizácia nám umožňuje správne vopred načítať stránky a dáta do cache pamäte. Ak *garbage collector* dokáže prispieť k zlepšeniu lokality dát, môže kladne ovplyvniť rýchlosť behu mutátora.

2.5 Metriky pre porovnávanie algoritmov automatickej správy pamäte

Porovnávať efektivitu jednotlivých prístupov je nesmierne náročné. Efektivita použitých algoritmov nezávisí len na objeme a štruktúre alokovaných objektov na halde, ale aj na prístupových vzoroch mutátora k alokovaným objektom. Ďalším problémom je, že stanovenie ideálnych parametrov pre algoritmy garbage collectingu pri jednej aplikácii môže viesť k nepriaznivému vplyvu na výkon pri inej aplikácii.

2.5.1 Bezpečnosť

Primárne hľadisko pri hodnotení GC je bezpečnosť. GC nesmie nikdy uvoľniť pamäť živých, dosiahnuteľných, objektov. Avšak bezpečnosť prináša istú cenu hlavne pre konkurenčné algoritmy (viac v kapitole 4). Bezpečnosť konzervatívnych algoritmov, ktoré nepotrebujú asistenciu prekladača alebo behového prostredia, môže byť narušená optimalizáciami prekladača, ktoré nejakým spôsobom menia referencie.

2.5.2 Priepustnosť

Základným cieľom pre užívateľa je, aby program bežal rýchlejšie. Avšak táto rýchlosť je ovplyvnená niekoľkými aspektami. Jeden z nich je, že celkový čas strávený *garbage collectingom,* by mal byť čo najkratší. Pomer behu *garbage collectora* a mutátora je často nazývaný *mark/cons ratio*. V najlepšom prípade strávi procesor maximum času vykonávaním efektívneho kódu mutátora a minimum času *garbage collectingom*. Preto je niekedy výhodné znížiť objem *garbage collectingu* za účelom zvýšiť procesorový čas mutátora. Na druhej strane niektoré *garbage collectory* môžu zvýšiť výkon mutátora. Napríklad *mark-compact* algoritmus (kapitola 3.2) po vykonaní zhutňovacej fázy zníži fragmentáciu pamäte a tým urýchli alokovanie nových objektov.

2.5.3 Úplnosť a promptnosť

Ideálne by mal GC uvoľniť všetky nedosiahnuteľné objekty ihneď, keď sa stane objekt nedosiahnuteľným. To však nie je vždy možné. Niekedy dokonca ani nie požadované. Napríklad čisté počítanie referencií (kapitola 3.4) nedokáže uvoľniť cyklické štruktúry. Kvôli požiadavkám na výkonnosť niekedy nie je požadované naraz zozbierať a uvoľniť nedosiahnuteľné objekty z celej haldy. Napríklad generačné *collectory* rozdeľujú objekty podľa ich veku na dve a viac skupín, nazvaných generáciami (kapitola 3.5). Častým uvoľňovaním mladých objektov a menej častým starších, sa zníži celkový procesorový čas GC a rovnako sa zníži aj priemerný čas pozastavenia mutátora pri jednotlivých cykloch GC.

2.5.4 Čas pozastavenia

Dôležitou požiadavkou na GC je minimalizovať pozastavenia mutátora jeho behom. Množstvo GC počas svojho behu pozastavujú všetky vlákna mutátora (*stop-the-world*). Samozrejme je požadované skrátiť tieto pozastavenia na čo najkratší čas. To môže byť kritické pri aplikáciách interagujúcich s užívateľom alebo pri aplikáciách, kde je potrebná okamžitá odpoveď (napríklad, kde nedodržanie *timeoutu* môže viesť k zrušeniu transakcie). Existujú rôzne prístupy ku skracovaniu až eliminácii tohoto času.

Generačné algoritmy zbierajú jednotlivé generácie s rôznou frekvenciou, čím redukujú potrebný čas pre ich beh. Paralelné algoritmy zas po zastavení mutátora využívajú viacero vláken pre uvoľňovanie pamäte a tým sa snažia skrátiť ich beh. Konkurenčné algoritmy využívajú vlákna, aby vykonávali svoju činnosť počas behu mutátora. To si zas vyžaduje synchronizáciu medzi mutátorom a GC, čo zvyšuje záťaž mutátora. Rovnako týmto prístupom narastá komplexnosť behového prostredia.

2.5.5 Priestorová zložitosť

Cieľom správy pamäte je zabezpečiť bezpečné a efektívne využitie pamäte. Rôzne prístupy k správe pamäte (jednak explicitné ale aj automatické) sa vyznačujú rôznou priestorovou zložitosťou. Niektoré GC môžu zvýšiť pamäť potrebnú pre uloženie každého objektu (napríklad uložením počtu referencií v objekte). Iné dokážu požadované informácie zakomponovať už do existujúcich častí objektu (*mark* bit môže byť uložený priamo v nevyužitých bitoch hlavičky objektu). S inými algoritmami zas prichádza zvýšenie pamäťovej náročnosti na celú haldu (kopírovacie algoritmy rozdeľujú haldu na minimálne dve časti, z ktorých vždy efektívne využívajú len jednu).

2.5.6 Optimálnosť pre jednotivé jazyky

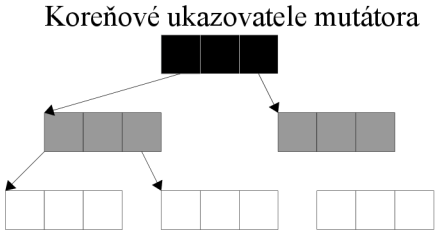
Algoritmy automatickej správy pamäte môžu byť charakteristické využiteľnosťou pre jednotlivé programovacie paradigmy. Generačné algoritmy môžu napríklad vyhovovať jazykom ako je Python, ktoré delia objekty na meniteľné a nemeniteľné. V deklaratívnych jazykoch sa zas pri backtrackingu stávajú všetky objekty vytvorené po bode návratu nedosiahnuteľnými a môžu byť uvoľnené v konštantnom čase. Niektoré jazyky zas podporujú deštruktory objektov, ktoré musia byť spustené pred uvoľnením objektu. Rôzne jazyky teda môžu vyžadovať rôzne vlastnosti GC.

2.5.7 Rozšíriteľnosť a prenositeľnosť

Poslednou metrikou je rozšíriteľnosť a prenositeľnosť. So zvyšujúcou sa tendenciou používania viacjadrových procesorov narastá potreba algoritmov, ktoré vedia naplno využiť tieto možnosti. Okrem toho očakávame, že v budúcnosti sa bude počet jadier zvyšovať a budú sa čoraz viac zavádzať heterogénne jadrá, optimalizované pre rôzne druhy aplikácií. Rastie aj veľkosť haldy, ktorú treba spravovať. Preto sú potrebné GC, ktoré sa dokážu vysporiadať s vyššie uvedenými faktami.

2.6 Trojfarebná abstrakcia

Pri popise algoritmov *garbage collectingu* je často využívaný model trojfarebnej abstrakcie. Využívajú ju hlavne trasovacie algoritmy. Pomocou nej popisujeme stav uzlov (objektov) pri prechádzaní grafom. Čierne uzly popisujú dosiahnuteľné (živé) objekty a biele potencionálne nedosiahnuteľné objekty. Zozačiatku sú všetky uzly biele. Pri prvom prechode uzlom, sa uzol zafarbí na sivo. Pokiaľ sú spracované všetky jeho deti (zafarbené na sivo alebo na čierno), uzol je prefarbený na čierno. Uzol je teda čierny, ak ho algoritmus spracoval a sivý, ak o ňom algoritmus vie, no nie je ukončené jeho spracovávanie. Obrázok 2.1 popisuje príklad tohoto algoritmu, uplatneného na objekty a koreňové ukazovatele mutátora.



Obrázok 2.1: Trojfarebná abstrakcia. Čierne objekty a ich deti boli GC už spracované. O sivých objektoch GC vie, no ešte ich nespracoval. K bielym objektom ešte nebola nájdená cesta referencií od koreňových ukazovateľov. Prebraté z [1].

3 Sekvenčné algoritmy automatickej správy pamäte

Táto kapitola popisuje a porovnáva sekvenčné verzie najznámejších algoritmov *garbage collectingu*, ktoré pozastavujú beh mutátora, tzv. *stop-the-world* algoritmy. Všetky algoritmy automatickej správy pamäte sú založené na jednom zo štyroch prístupov. Sú to *mark-sweep*, kopírovanie, *mark-compact* a počítanie referencií. Delia sa na nepriame, ktoré hľadajú všetky dosiahnuteľné objekty a ostatné pokladajú za nedosiahnuteľné a priame, ktoré priamo vyhľadajú nedosiahnuteľné objekty.

3.1 Mark-sweep algoritmus

Mark-sweep algoritmus patrí medzi trasovacie nepriame algoritmy.

3.1.1 Princíp algoritmu

Tento algoritmus vychádza priamo z rekurzívnej definície dosiahnuteľnosti objektov cez ukazovatele. Má dve fázy. V tzv. *marking phase* (označovacia fáza) prechádza graf objektov začínajúci v koreňových ukazovateľoch mutátora. Každý nájdený objekt označí. Označením môžeme chápať napríklad nastavenie vybraného bitu v objekte, alebo v tabuľke. Tento prechod grafom sa tiež nazýva trasovanie. V druhej fáze, ktorá je nazvaná *sweeping phase* (upratovacia fáza), *garbage collector* prechádza všetky objekty na halde. Každý neoznačený objekt nie je dosiahnuteľný mutátorom, a teda môže byť uvoľnený. Príklad implementácie mark-sweep algoritmu sa nachádza v prílohe A.1.

3.1.2 Hodnotenie algoritmu

Tento algoritmus má zlú priestorovú a časovú lokalitu. V označovacej fáze GC prechádza väčšinou objektov len raz, keďže väčšina objektov nie je zdieľaná. Ak teda bol objekt označený, tak je veľmi nepravdepodobné, aby sa k nemu pristúpilo v tejto fáze opäť. Existuje niekoľko vylepšení, ktorými sa dá táto zlá lokalita kompenzovať.

Mark-sweep algoritmus má ale niekoľko výhod. Často sa preto používa ako základný algoritmus pre sofistikovanejšie metódy. Mark-sweep nezvyšuje zaťaženie mutátora režijnými operáciami. Ponúka dobrú priepustnosť, keďže označovacia fáza je relatívne rýchla a jej trvanie je ovplyvnené hlavne rýchlosťou prístupu cez ukazovatele. V objekte nastavuje iba jeden bit a nepresúva alebo nekopíruje dané objekty. Potreba jedného bitu v objekte nemusí zvýšiť pamäťové nároky behu programu.

Na druhej strane pozastavenie mutátora je silno ovplyvnené typom mutátora. V najhorších prípadoch, pri veľmi komplexných systémoch, môže pozastavenie trvať aj niekoľko sekúnd. Ďalším negatívom je, že neprispieva k zlepšeniu lokality dát pre mutátor a k zníženiu fragmentácie haldy. To negatívne ovplyvňuje jeho využívanie priestoru a sám touto vysokou fragmentáciou trpí. Preto potrebuje komplexnejšie riešený alokátor, ktorý na rozdiel od zhutňujúcich alebo kopírujúcich GC dokáže vyhľadať optimálne miesto pre alokovanie nového objektu.

3.2 Mark-compact algoritmy

Táto skupina algoritmov sa snaží riešiť fragmentáciu haldy. Zníženie fragmentácie pomáha lepšej lokalite dát a uľahčuje prácu alokátora pri hľadaní voľného miesta. Opäť sa jedná o trasovacie nepriame algoritmy.

3.2.1 Princíp algoritmu

Mark-compact algoritmy pracujú v niekoľkých fázach. Prvá fáza je zhodná s označovacou fázou mark-sweep algoritmu. V nasledujúcich zhutňujúcich fázach tieto algoritmy fyzicky presúvajú v pamäti objekty tak, aby zaberali čo najsúvislejšie miesto v pamäti a následne aktualizujú hodnoty všetkých referencií živých objektov. Počet a poradie týchto fáz sa líšia od konkrétneho algoritmu. Objekty môžu byť na halde presúvané troma spôsobmi.

- **Náhodne** – Objekty sú presunuté bez akýchkoľvek zreteľov na ich pôvodnú polohu alebo na ich vzťah k iným objektom cez referencie.
- **Linearizáciou** – Objekty sú presunuté tak, aby objekty na seba ukazujúce v aspoň jednom smere, v pamäti fyzicky susedili, resp. aby susedili objekty, ktoré sú súrodencami v istej dátovej štruktúre.
- **Stlačením** – Objekty sú stlačené na jednu stranu haldy, a tým vytlačia von nedosiahnuteľné objekty. Ich relatívna poloha voči sebe je teda zhodná s ich relatívnou polohou pred behom GC.

Väčšina GC využíva náhodné presúvanie alebo stláčanie. Náhodné presúvanie je jednoduché na implementáciu a rýchle na beh GC, hlavne ak sú všetky objekty rovnakej veľkosti. Pri rozličnej veľkosti objektov sa musia spravovať skupiny objektov so zhodnou veľkosťou oddelene. Náhodné presúvanie ale vedie ku zlej priestorovej lokalite pre mutátor, pretože objekty, ktoré spolu nejakým spôsobom súvisia, sa môžu ocitnúť v rôznych častiach pamäte. Moderné GC preto tento spôsob nepoužívajú. Často tieto zhutňujúce algoritmy potrebujú extra slot v hlavičke objektov na uloženie ich budúcej polohy. Príklad takéhoto algoritmu je uvedený v prílohe A.2.

3.2.2 Hodnotenie algoritmov

Táto rodina algoritmov má približne rovnako nízku pamäťovú náročnosť ako *mark-sweep*, čo je približne o polovicu menej, ako je potrebné pre kopírovacie algoritmy. Na rozdiel od mark-sweep algoritmu, zhutňujúce algoritmy dokážu efektívne riešiť problém fragmentácie, a tým sú vhodné pre dlho bežiace aplikácie. Čo sa týka priepustnosti, oproti *mark-sweep* algoritmu, zrýchľujú operáciu alokácie. Sú vhodné pre vytvorenie paralelných verzií.

Samozrejme, je tu aj druhá stránka veci. Trvanie cyklov týchto algoritmov je väčšie ako trvanie cyklu *mark-sweep* algoritmu. Mnohé z týchto algoritmov kladú aj väčšiu režijnú záťaž alebo obmedzenia (ako používať objekty rovnakej veľkosti) na mutátor. Globálne ponúkajú zhutňujúce algoritmy horšiu priepustnosť ako *mark-sweep* algoritmy, pretože potrebujú viac prechodov haldou. Tieto prechody sú časovo náročné, keďže sú spojené s častým prístupom cez ukazovatele a prechádzaním množstva objektov v rôznych častiach pamäte. Sú ale dobre využiteľné pri kombinovaní s *mark-sweep* algoritmom, kde sa vkladajú zhutňujúce cykly medzi cykly *mark-sweep* algoritmu.

Zhutňujúce GC si dokážu efektívne poradiť aj s dlho žijúcimi objektami. Na rozdiel od kopírovacích algoritmov, ktoré tieto objekty kopírujú stále z jednej časti haldy do druhej, *mark-*

compact algoritmy si tieto objekty dokážu držať na začiatku haldy, a tým minimalizovať ich presúvanie. Síce použitím náhodného presúvania sa zhoršuje lokalizácia objektov, použitím linearizácie alebo stlačenia môžu tieto algoritmy lokalizáciu mutátora zlepšiť, a tým zrýchliť jeho beh.

3.3 Kopírovacie algoritmy

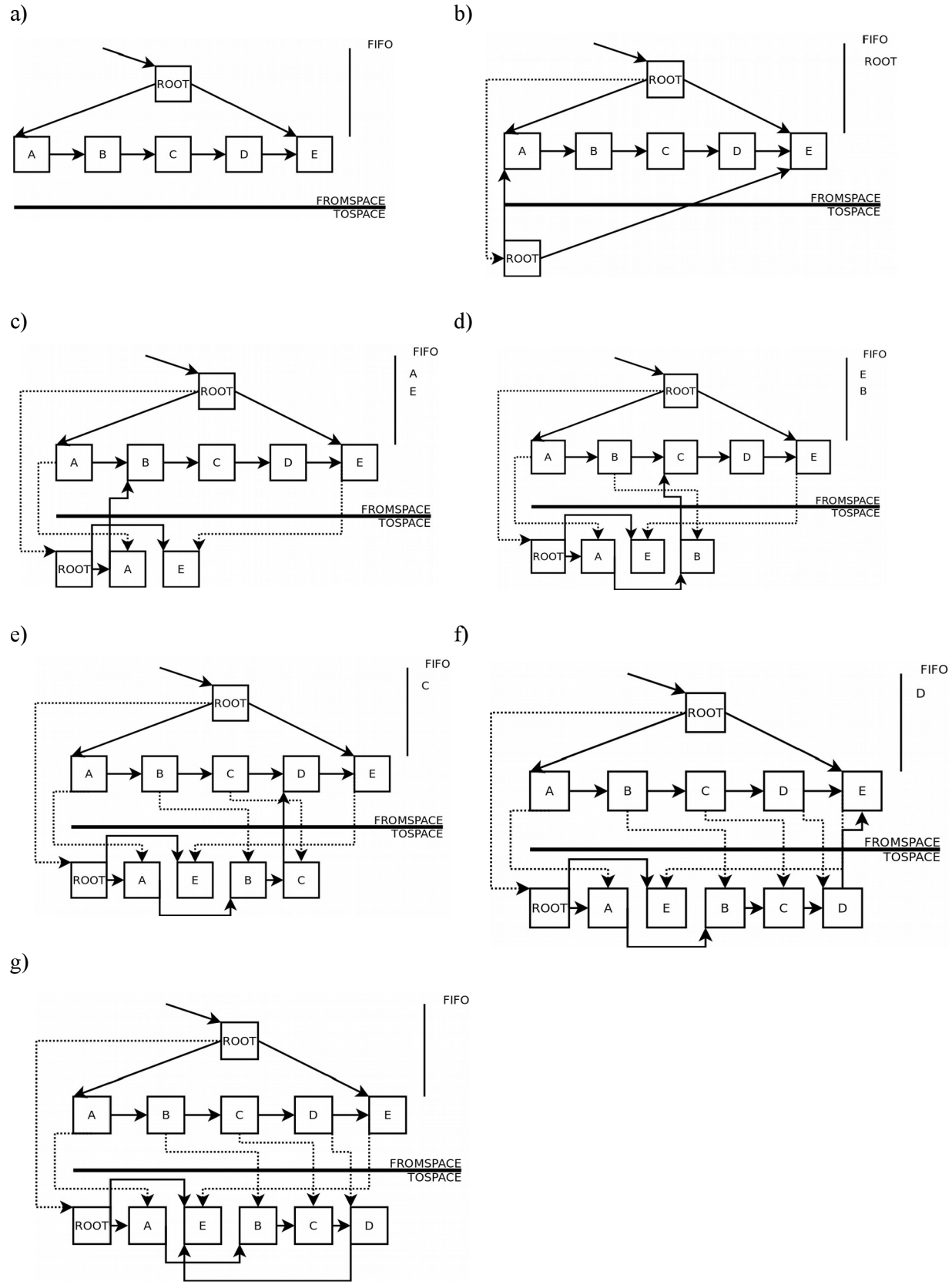
Kopírovacie algoritmy ponúkajú rýchlu alokáciu z dôvodu odstránenia fragmentácie, a to jediným prechodom haldy. Aj táto rodina algoritmov sa radí medzi nepriame.

3.3.1 Princíp algoritmu

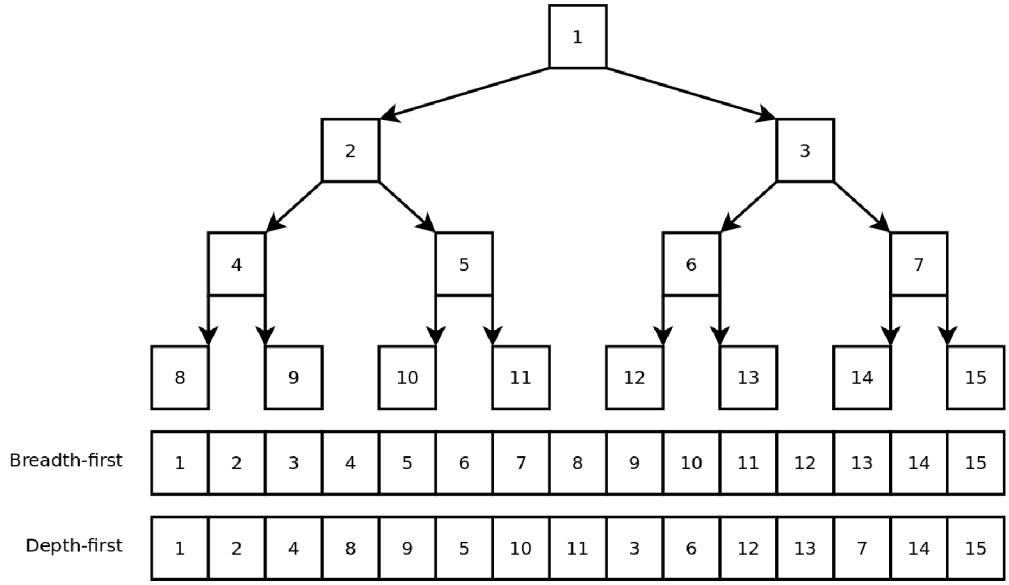
Základné kopírovacie GC delia haldu na dve rovnako veľké časti, ktoré sa volajú *fromspace* a *tospace*. Základný algoritmus je uvedený v prílohe A.3. Predpokladá síce spojitosť haldy, no s menšími úpravami je ho možné použiť aj pre haldu, ktorá nie je spojitá (napriklad pre časti alokované operačným systémom). Mutátor vždy používa len jednu časť haldy, a to *tospace*. Nové objekty sú alokované, ak je tam dostatok priestoru, v *tospace* zvýšením ukazovateľa ukazujúceho na prvé voľné miesto (*free* ukazovateľ). V opačnom prípade sa *fromspace* a *tospace* vymenia a živé objekty sú prekopírované z *fromspace* (bývalého *tospace*) do *tospace* (bývalého *fromspace*). Tým sa eliminuje fragmentácia a odstránia sa nedosiahnuteľné objekty z používanej časti haldy.

Cyklus sa začína skopírovaním koreňových objektov do *tospace*. Skopírované, no ešte neprehľadané objekty, sú podľa trojfarebnej abstrakcie označené sivo. Každý atribút, ktorý je ukazovateľ v sivom objekte, má hodnotu buď *null* alebo referenciu na objekt vo *fromspace*. Následne sú prechádzané všetky ukazovatele sivých objektov, ktoré sú aktualizované na hodnotu adresy kópií objektov, na ktoré pôvodne ukazovali v *tospace*. Ak pri prechode objektami algoritmus narazí na objekt vo *fromspace*, zistí, či už bol skopírovaný do *tospace*. Ak nebol, tento objekt skopíruje na miesto, na ktoré ukazuje ukazovateľ *free*. Ten je následne inkrementovaný o veľkosť skopírovaného objektu. Základnou vlastnosťou tohoto presunu objektov je, že po skopírovaní do *tospace* zachovávajú pôvodnú topológiu uloženia objektov vo *fromspace*. Cyklus sa končí, ak boli prezreté všetky objekty v *tospace*.

Na rozdiel od *mark-compact* algoritmov, kopírovacie algoritmy nepotrebujú žiadne miesto v hlavičke objektov. Akékoľvek miesto vo *fromspace* môže byť použité na uloženie adries pre presun objektov. To umožňuje použiť tento algoritmus pre objekty bez hlavičiek. Podobne, ako aj trasovacie algoritmy, kopírovacie algoritmy potrebujú zoznam spracovávaných objektov. Obrázok 3.1 popisuje príklad kopírovacieho algoritmu používajúceho FIFO frontu (prechod grafom objektov do šírky – breadth-first) a obrázok 3.2 popisuje rozdiel uloženia objektov v tospace na základe použitia FIFO resp. LIFO fronty pri referenčnom grafe objektov.



Obrázok 3.1: Obrázky ukazujúce jednotlivé kroky kopírovacieho algoritmu. Prerušované šípky znázorňujú uloženie referencie s adresou kópie objektu v *tospace*.



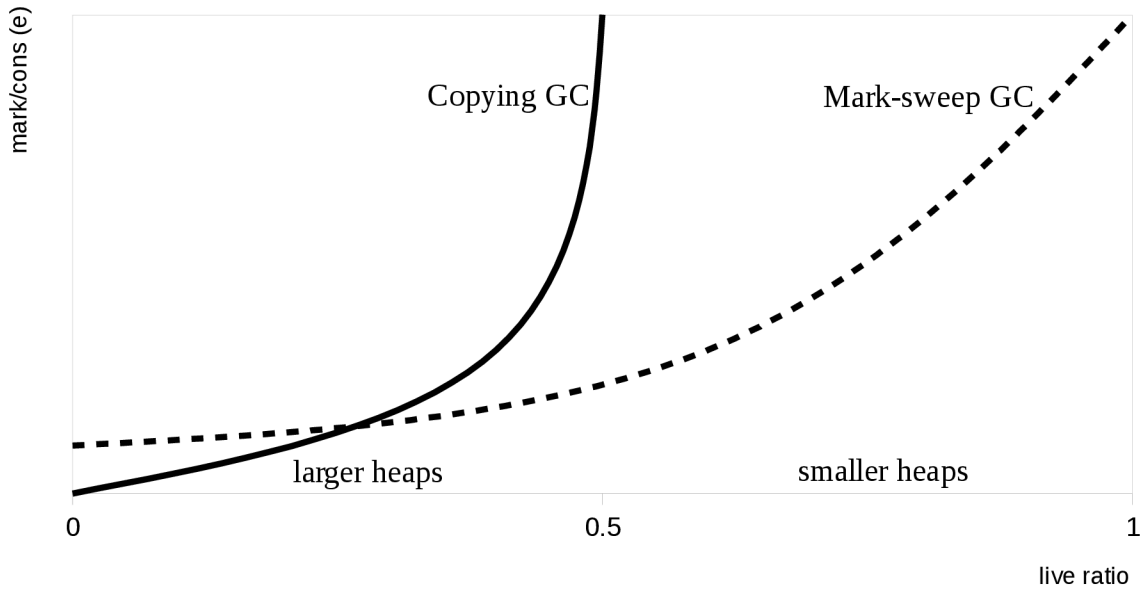
Obrázok 3.2: Obrázok znázorňujúci uloženie objektov z grafu objektov v tospace v závislosti od použitého algoritmu vyberania objektov z fronty.

3.3.2 Hodnotenie algoritmu

Kopírovacie algoritmy ponúkajú oproti *mark-sweep* dve výhody. Sú to rýchla alokácia a odstránenie fragmentácie. Alokácia je rýchla, pretože je jednoduchá. Bežne si vyžaduje len test na dostatok miesta od oblasti, kde ukazuje free ukazovateľ po koniec tospace. Rovnako jednoducho je riešená aj pre paralelné viacvláknové programy. Každému vláknu je priradený alokačný buffer, v ktorom pracuje bez potreby synchronizácie.

Nevýhodou kopírovacích algoritmov je potreba udržiavať druhú, nepoužívanú, časť haldy. Na rozdiel od ostatných algoritmov teda ponúka pre využitie len polovicu veľkosti haldy, a teda musí častejšie vykonávať cykly. Môže sa teda zdať, že mark-sweep je efektívnejší. [1] ale uvádza, že to platí iba pre malé haldy. Pri dostatku miesta sekvenčná alokácia objektov, a tým menší počet cache missov, prevažuje nad častými cyklami GC. Graf 3.1 znázorňuje porovnanie efektivity *mark-sweep* a kopírovacích algoritmov. To, či to zníži a o koľko to zníži efektívny beh mutátora oproti iným algoritmom, teda závisí nie len od charakteru aplikácie, ale aj od veľkosti miesta na halde.

Je zrejmé, že kopírovacie algoritmy môžu meniť relatívne uloženie objektov medzi sebou a tým zlepšiť priestorovú lokalitu pre beh mutátora. Nanešťastie podľa [1] sú dva dôvody, prečo nie je možné nájsť optimálne uloženie objektov pre efektívne využitie vyrovnávacích pamätí. GC nemôže dopredu poznať vzor prístupov mutátora k objektom. Horší je ale druhý dôvod, ktorý hovorí, že problém uloženia objektov v pamäti je NP-úplný. Čiže aj napriek prípadnej znalosti prístupových vzorov mutátora k objektom, neexistuje efektívny algoritmus pre výpočet optimálneho uloženia objektov.



Graf 3.1: Graf porovnania efektivity *mark-sweep* a kopírovacieho GC. *Mark / cons (e)* označuje veľkosť práce, ktorú je potrebné vykonať na alokovanú jednotku. Mark sweep musí vždy prejsť vo *sweep* fáze zoznamom všetkých objektov, preto aj pri krátko žijúcich objektoch vykoná na alokovanú jednotku isté množstvo práce. Kopírovacie algoritmy zas dlho žijúce objekty veľakrát kopírujú z *fromspace* do *tospace*, čo je dosť drahá operácia a tým značne narastá veľkosť práce na alokovanú jednotku. Pre väčšie haldy je potrebné menej časť spúšťať cyklus GC, preto vek objektov sa zvyšuje pomalšie, ako pre menšie haldy. Prebraté z [1].

3.4 Algoritmy založené na počítaní referencií

Všetky vyššie uvedené algoritmy boli nepriame. Hľadali živé objekty, a ktorý objekt nebol živý, považovali za nedosiahnuteľný. Počítanie referencií je priama metóda.

3.4.1 Princíp algoritmu

Tento algoritmus je založený na myšlienke, že objekt je nedosiahnuteľný vtedy, ak je počet referencií naň ukazujúci nulový. Počítanie referencií je preto späté s informáciou o počte referencií ukazujúcich na každý objekt. Táto informácia je uložená v hlavičke objektu. V najjednoduchšej implementácií, ktorá je v prílohe A.4, sa počítadlo referencií inkrementuje a dekrementuje vtedy, keď je vytvorená alebo zrušená referencia na objekt. Ak počet referencií na objekt klesne na nulu, objekt môže byť uvoľnený. To dekrementuje počty referencií na všetky jeho synovské objekty, kde sa musí tento proces opakovať.

3.4.2 Hodnotenie algoritmu

Existuje mnoho výhod tohoto algoritmu. Réžia spojená so správou pamäte je rozptýlená cez beh celého programu. To pozitívne vplýva na v priemere krátke pauzy mutátora. Potencionálne dokáže tento algoritmus uvoľniť objekty hneď, ako sa stanú nedosiahnuteľné. Dokáže pracovať aj v relatívne plnej halde, s čím majú trasovacie GC výkonnostné problémy. Lokalizácia tohoto algoritmu je tiež

veľmi dobrá, keďže priamo pracuje so zdrojovým a cieľovým objektom pre danú referenciu. To zabezpečuje minimálne rovnako dobrú lokalizáciu ako lokalizácia mutátora. Počítanie referencií môže byť implementované aj bez globálnej znalosti o objektoch a znalosti behového prostredia, čo je výhoda pri distribuovaných systémoch.

Na druhej strane počítanie referencií má množstvo značných nevýhod. S inkrementáciou a dekrementáciou počítadiel referencií sa kladie vyššia záťaž na mutátor. Obe operácie, vytvorenie aj rušenie referencie, musia byť atomické. To je dôležité hlavne pri viacvláknových programoch. Z tohoto dôvodu je potrebné implementovať čítaciu a zápisovú bariéru. Najväčšou nevýhodou tohoto algoritmu je, že nedokáže uvoľňovať cyklické štruktúry. Ak je takýto cyklus objektov izolovaný od koreňa mutátora, teda je nedosiahnuteľný, tento algoritmus to v jeho čistej forme nedokáže detekovať, pretože počet referencií v cykle bude stále nenulový pre každý objekt v cykle. Ďalšou nevýhodou je, že počet referencií na objekt môže byť veľký ako počet všetkých objektov na halde. To znamená, že veľkosť každého objektu sa zväčší o veľkosť ukazovateľa (pridanie počítadla referencií do hlavičky), čo je pri 64 bitových systémoch práve 64 bitov. Poslednou nevýhodou je, že pri uvoľňovaní veľkých objektov sa musia prechádzať a uvoľňovať aj jeho synovské objekty, a to môže spôsobiť dokonca dlhšie pauzy ako pri trasovacích GC.

3.5 Generačný prístup

Cieľom GC je nájsť nedosiahnuteľné objekty a uvoľniť miesto, ktoré zaberajú. Trasovacie a kopírovacie GC sú najefektívnejšie, ak halda obsahuje málo živých objektov. Naopak, dlho žijúce objekty spomaľujú tieto algoritmy, keďže sú opakovane prechádzané, resp. kopírované. V predošlej podkapitole je uvedené, že s týmito objektami dokážu efektívne pracovať zhutňovacie GC, ktoré ich premiestnia na začiatok haldy.

Generačné GC sa odrážajú práve od týchto myšlienok tým, že staršie objekty sú menej často prechádzané a môžu byť prechádzané aj inými algoritmami ako mladšie, ktoré sa častejšie stávajú nedosiahnuteľnými. Tým sa snažia maximalizovať uvoľnené miesto pri minimálnej námahe. Generačné GC sú schopné zvýšiť priepustnosť, pretože použité algoritmy neprechádzajú všetky objekty.

Negatívom je, že objekty zo staršej generácie, ktoré sa stanú nedosiahnuteľnými, nie sú uvoľnené pri *garbage collectingu* mladšej generácie. To znamená, že uvoľňovanie starších objektov nie je okamžité. Na to, aby boli generačné GC schopné zozbierať len jednu generáciu, potrebujú istú réžiu naviac, ktorá oddelí mladšie objekty od starších.

3.5.1 Problém správneho načasovania presunu do staršej generácie

Pri generačných GC je potrebné stanoviť správny časový parameter, po ktorom sú objekty presunuté do staršej generácie. Ak je tento čas pridlhý, prehľadáva sa v mladšej generácii priveľa objektov. Keďže mladá generácia podlieha častému behu GC, priveľa objektov v mladej generácii výrazne vplýva na výkon. Ak je však priskoro objekt presunutý do staršej generácie, táto generácia sa rýchlo zaplní a aj pre ňu by bol potrebný častý beh GC. Tým sa stráca význam generačného prístupu. Preto je potrebné nájsť optimálne parametre pre dané behové prostredie, vlastnosti jazyka a aplikácie, ktoré v danom prostredí bežia. Často je tento problém riešený adaptatívnym časovaním automaticky prispôsobiteľným pre danú aplikáciu.

3.5.2 Rôzne prístupy pri presúvaní objektov do starších generácií

Objekty môžu byť presúvané do staršej generácie buď po jednom alebo skupinovo v celku. Výhodou prvého prístupu je, že do starších generácií sa dostávajú len objekty, ktoré tam reálne patria. Na druhej strane takýto prístup si vyžaduje informáciu o čase vytvorenia daného objektu. Druhá možnosť je presúvať objekty skupinovo. Tu je potrebné pamätať si len vek danej skupiny. Problém tohto prístupu je, že do staršej generácie sa so skupinou môžu dostať aj objekty, ktoré sa do danej skupiny dostali len nedávno, a tak majú vyššiu tendenciu stať sa nedosiahnuteľnými, a do staršej generácie nepatria.

Presun do staršej generácie môže byť jednoduchý použitím kopírovacích algoritmov pre správu mladšej generácie. Mladšia generácia môže byť niekoľkokrát zozbieraná, kým sa všetky jej živé objekty presunú do staršej. Napríklad objekty mladšej generácie sú niekoľkokrát kopírované z *fromspace* do *tospace*, no stále v rámci jednej generácie. Pri ďalšom cykle sú skupinovo skopírované z nového *fromspace* do staršej generácie. Napriek tomu, že tento prístup správne presunie staršie objekty do staršej generácie, presunie s nimi aj množstvo mladých objektov, ktoré sú presunuté predčasne.

Zjemnením skupín je možné dosiahnuť lepšie výsledky, a to v prípade, že je jedna generácia rozdelená na viacero skupín a pri každom cykle GC sú presúvané objekty z jednej skupiny do druhej a z najstaršej skupiny do staršej generácie. To zabezpečí, že objekt musí prežiť minimálne n cyklov pri n skupinách, aby bol presunutý do staršej generácie. Tento prístup zas zvyšuje náročnosť kopírovacieho GC.

3.5.3 Hodnotenie algoritmov

Generačné algoritmy sa osvedčili ako efektívny spôsob automatickej správy pamäte. Ponúkajú značné zvýšenie výkonnosti pre rozsiahlu množinu aplikácií. Ohraničením veľkosti pamäte pre mladú generáciu a jej frekventovaným spracovávaním dokážu výrazne zredukovať pozastavenia mutátora. Tento prístup dokáže zvýšiť celkovú priepustnosť dvoma spôsobmi. Prvý je, že opakovane nespracováva dlho žijúce objekty, čím im dáva aj viac času na to, aby sa stali nedosiahnuteľnými. Druhý sa týka sekvenčného alokovania nových objektov v časti pamäte, ktorá je na to určená. To umožňuje dobrú priestorovú aj časovú lokalitu, pretože najviac operácií sa deje s mladými objektami.

Avšak výkon týchto algoritmov je silno závislý na nastavených parametroch a na druhu aplikácie. Algoritmus nesmie presúvať objekty do vyššej generácie ani príliš skoro, ani príliš neskoro. Čas strávený frekventovaným prechádzaním mladých objektov musí byť samozrejme kompenzovaný ziskom pri neprechádzaní starších. Ak aplikácia obsahuje málo objektov, ktoré sa stávajú nedosiahnuteľnými zamlada, generačné algoritmy nebudú tou správnou voľbou. Po ďalšie, generačné algoritmy zlepšia iba priemerný čas pozastavenia mutátora. Niekedy musí byť zozbieraná celá halda a generačný algoritmus nedokáže zabezpečiť zredukovanie maximálneho času pozastavenia. Síce je jednoduchšie implementovať generačné algoritmy, ak GC je schopný hýbať objektami, a tým vytvárať fyzické časti haldy s mladšími a staršími objektami, ale tento spôsob môže značne narušiť dobrú lokalitu dát. Samozrejme, objekty môžu byť rozdelené aj logicky, napríklad nastavením bitu v ich hlavičke.

4 Paralelné a konkurenčné algoritmy GC

Dnešný trend vo vývoji hardvéru je zvyšovať počet procesorov a jadier. Tým vzrastá priestor pre uplatnenie paralelných algoritmov na klasických desktopových počítačoch alebo notebookoch. V predchádzajúcej kapitole som uviedol sekvenčné verzie algoritmov použiteľných pri *garbage collectingu*. Uvažoval som jedno alebo viac vláken mutátora, ktoré následne sú pozastavené jediným vláknom GC. To je značné nevyužitie ponúkaných výpočtových zdrojov danej architektúry. To viedlo k vytvoreniu paralelných a konkurenčných algoritmov pre GC, ktoré popisujem v tejto kapitole. Je dôležité spomenúť rozdiel medzi paralelným a konkurenčným GC. Paralelný znamená, že *garbage collector* využíva viacero vláken. Konkurenčný zas implikuje, že GC nepozastavuje beh mutátora, ale využíva iné vlákno na svoj vlastný beh. Tieto prístupy samozrejme ide kombinovať, a tak vytvoriť konkurenčný paralelný GC.

4.1 Paralelné algoritmy

Pred rozhodnutím pre paralelizmus je potrebné si zodpovedať niekoľko otázok. Paralelné algoritmy často potrebujú istú réžiu pre synchronizáciu vláken. Preto je potrebné, aby pre paralelné vykonávanie bolo dostatok práce, ktorá sa dá rozdeliť do čo najviac nezávislých úloh. V opačnom prípade vďaka réžii môže byť paralelné vykonávanie algoritmu pomalšie ako jeho sekvenčná podoba.

Druhým problémom je vyvažovanie záťaže jednotlivých vláken. Hlavnou myšlienkou paralelizmu je distribuovať prácu medzi jednotlivé zdroje ponúkané architektúrou. Ideálny stav je, ak všetky ponúkané zdroje sú zaťažené rovnako, a teda sú plne využité. Bez vyvažovania záťaže môže dôjsť k preťaženiu istého zdroja, kým ostatné zdroje sú nevyužité a to môže viesť len k minimálnemu urýchleniu oproti sekvenčnému vykonávaniu. Existujú dva prístupy, a to je statické a dynamické balansovanie. Statické balansovanie rozdelí záťaž v dobe prekladu aplikácie, a tým za jej behu nepotrebuje žiadnu, resp. potrebuje len minimálnu synchronizáciu a komunikáciu medzi vláknami. Napríklad paralelná verzia kopírovacích algoritmov rozdelí *fromspace* a *tospace* na časti. Každé vlákno sa stará len o jednu svoju časť. Na druhej strane to nemusí vždy viesť k rovnomernému rozloženiu záťaže medzi všetky zdroje. Dynamické rozdelenie záťaže zas túto záťaž dokáže rozdeľovať rovnomernejšie. Cenou za to je zvýšenie komunikácie a synchronizácie medzi jednotlivými zdrojmi. Napríklad paralelná verzia *mark-compact* algoritmu po označovacej fáze identifikuje jednotlivé živé objekty. Tie následne rovnakým dielom rozdelí medzi jednotlivé vlákna tak, aby boli paralelne spracované.

Častokrát nie je možné stanoviť dopredu ani v čase behu aplikácie množstvo práce, ktorú treba vykonať. Vtedy sa práca delí na viacero menších častí. Tento prístup má viacero výhod. Nie je náchylný na zmenu počtu zdrojov, keďže menšie úlohy môžu byť ľahšie rozdelené medzi rôzny počet procesorov. Ak nejakému vláknu trvá jeho úloha dlhšie, neovplyvní tým vyváženosť vykonávania daného algoritmu.

Ďalšie rozdelenie paralelných algoritmov je na procesocentrické a pamäťocentrické. Procesocentrické rozdeľujú algoritmus na rôzne veľké podúlohy. Nekladú dôraz na lokalitu dát, s ktorými pracujú. Pamäťocentrické na druhej strane dbajú viac na pamäťovú lokalitu dát. Typicky

rozdeľujú pamäť na niekoľko rovnakých častí, ktoré následne tvoria jednotlivé podúlohy. Najčastejšie sú používané pri kopírovacích algoritmoch.

4.1.1 Paralelizovanie označovacej (marking) fázy

Označovacia fáza zahŕňa tri úlohy. Získanie objektu (podúlohy) pre vlákno zo zoznamu podúloh, testovanie a nastavenie jednej alebo viac značiek a generovanie nových podúloh do zoznamu podúloh. Všetky známe algoritmy na paralelizovanie označovacej fázy sú procesocentrické. Ak docielime, aby zoznamy podúloh boli prístupné len pre jedno jadro a zároveň boli neprázdne, tak nie je potrebná žiadna synchronizácia. V opačnom prípade musia vlákna získavať podúlohy atomickými operáciami. Ak nie je zabezpečená atomicita, môže sa stať, že synovské objekty objektu, ktorý je spracovávaný naraz viacerými vláknami, budú viackrát vložené do zoznamu podúloh, čím sa zníži rýchlosť algoritmu.

Pre dynamické rozdeľovanie podúloh sa dá použiť takzvaný *work stealing* prístup. Každé vlákno má svoj vlastný zoznam podúloh, s ktorými pracuje a kde vkladá novo vzniknuté podúlohy. V prípade, že tento zoznam vyprázdni, zoberie si podúlohu od iného vlákna. Samozrejme, že je tu potrebná synchronizácia a zabezpečenie atomicity pri braní podúlohy od iného vlákna, pretože viacero vláken sa môže pokúsiť zobrať tú istú podúlohu naraz. Celý algoritmus je uvedený v prílohe A.5.

4.1.2 Paralelné kopírovanie

Paralelné kopírovanie má podobné vlastnosti a problémy ako paralelná označovacia fáza. S výnimkou, ak je objekt viackrát označený, je to zanedbateľné, no pri kopírovaní musí byť každý živý objekt skopírovaný práve jedenkrát.

Rozloženie práce je realizované buď procesocentrickým alebo pamäťocentrickým prístupom. Pri procesocentrickom prístupe je každému kopírovaciemu vláknu priradený vlastný zásobník pre ukladanie podúloh. Záťaž je balansovaná tak, že vlákna periodicky prenášajú podúlohy medzi ich lokálnymi zásobníkmi a globálne zdieľaným zásobníkom. Je potrebné, aby globálny zásobník zabezpečoval výlučný prístup. Zdroj [1] odporúča využitie zásobníka ako dátovej štruktúry, keďže sa ľahko synchronizuje pri viacvláknovom prístupe. Napriek tomu, že zásobník nedovoľuje náhodný prístup k jeho položkám a pri požadovanom výlučnom prístupe by bol podstatným článkom s čiste sekvenčným prístupom, existuje možnosť, ako povoliť viacerým vláknam vkladať a vyberať položky zásobníka paralelne. Jediné, čo je potrebné, aby vlákno posunulo ukazovateľ na vrchol zásobníka, a tým si vytvorilo miesto, kde môže zapisovať alebo z ktorého môže čítať bez rizika *data race*. Celá implementácia takéhoto zdieľaného zásobníka je v prílohe A.6. Príloha A.6 obsahuje algoritmus na paralelné kopírovanie. Jediný problém tohto prístupu je, že neumožňuje miešať operácie vyberania a vkladania do zásobníka.

Na to, aby sa zabezpečilo, že iba jedno vlákno skopíruje objekt, je potrebné, aby prvé vlákno, ktoré k objektu pristúpi naznačilo, že daný objekt sa kopíruje. To sa dá zaistiť vložením adresy, kde sa objekt kopíruje do jeho hlavičky. To, ako vlákna kopírujú objekt, závisí od toho, či sú im pridelené jednotlivé nezdieľané pamäťové miesta alebo kopírujú do zdieľanej pamäte. Pri zdieľanom mieste je potrebné zaistiť atomicitu alokácie miesta pre kopírovaný objekt.

Pri paralelnom kopírovaní je možné použiť aj pamäťocentrický prístup. Najjednoduchšia možnosť je rozdeliť *fromspace* a *tospace* na toľko častí, koľko je vláken GC. Každé vlákno kopíruje

objekty z jemu pridelenej časti *fromspace* do jemu pridelenej časti *tospace*. To ale nezabezpečuje dostatočné vyváženie práce medzi vláknami.

Riešením môže byť vytvorenie viacerých takýchto častí *fromspace* a *tospace* a ich označenie za podúlohy. Vlákna si tak vyberajú z globálneho skladu jednotlivé časti *fromspace* a *tospace*. Ak naplnia svoju časť *tospace*, vrátia ju do skladu s plnými časťami *tospace* a vyberú si zo skladu s prázdnymi časťami *tospace* ďalšiu časť. Problémom tohto prístupu je fragmentácia na konci naplnených častí *tospace*.

4.1.3 Paralelizovanie sweep fázy

Paralelizovanie tejto fázy je celkom priamočiare. Dá sa to urobiť viacerými spôsobmi. Opäť je najjednoduchšia implementácia statické rozdelenie haldy na rovnako veľké časti. Ich počet môže byť rovnaký alebo väčší ako počet dostupných vláken. V druhom prípade je opäť potrebné zaviesť zoznam s danými podúlohami. To samozrejme robí z tohto zoznamu spomaľujúce miesto algoritmu. Dá sa to sčasti kompenzovať balíkmi podúloh, ktoré si jednotlivé vlákna vyberajú zo zoznamu. Následne vracajú balíky nových úloh a balíky vykonaných úloh do skladu vykonaných úloh.

4.1.4 Paralelizovanie compact fázy

Zhutňovanie má väčšinu problémov uvedených vyššie. Pri *mark-compact* algoritme musia byť objekty paralelne označené a paralelne presunuté. Paralelné stlačenie je o niečo jednoduchšie ako paralelné kopírovanie. Po označení je už jasná adresa, kde sa bude daný objekt premiestňovať. Teda je potrebné riešiť len rezervovanie daného objektu jedným vláknom.

Je možné rozdeliť *mark-compact* na tri kroky a každý z nich sekvenčne za sebou vykonávať paralelne. V prvom kroku sa paralelne označia živé objekty. V druhom sa paralelne vypočítajú adresy ich nového umiestnenia a v treťom sa paralelne aktualizujú ich ukazovatele. Je možné využiť aj rôzne stratégie balansovania záťaže pre každý krok.

Pri paralelnom stláčaní vzniká problém, že jedno vlákno môže prepísať nepresunutý objekt iného vlákna. Tento problém sa dá riešiť rozdelením haldy na viacero častí, pre každé vlákno jednou časťou, a jednotlivé vlákna potom stlačia presúvané objekty na začiatok ich častí. To samozrejme zapríčiňuje miernu fragmentáciu.

4.1.5 Hodnotenie algoritmu

Pred použitím paralelných algoritmov je potrebné si zodpovedať otázku, či je dostatok práce, ktorá by sa dala paralelizovať. Našťastie, moderné aplikácie využívajú širokú škálu rôznych dátových štruktúr, čo prispieva k možnosti paralelizácie. Algoritmy pre GC, s výnimkou trasovania, ponúkajú možnosť využiť paralelný hardvér. Napríklad *sweep* fáza alebo zhutňovanie sú triviálne paralelizovateľné. Dokonca aj pre trasovacie algoritmy po menších úpravách existujú paralelné verzie. Samozrejme cenou za paralelizáciu je réžia spätá so synchronizáciou a výlučným prístupom.

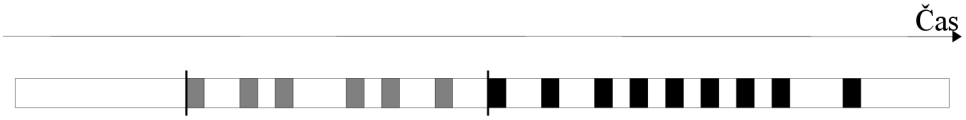
Ďalším problémom je zabezpečenie rovnomerného rozdelenia záťaže medzi všetky zdroje ponúkané hardvérom. Spravidla ide o procesory, resp. vlákna, ktoré dokážu bežať paralelne. Tu je potrebné nájsť vyváženosť medzi potrebnou réžiou synchronizácie pri rozdeľovaní záťaže a precíznosťou v rovnomernom rozdelení záťaže. Osvedčené riešenie je rozdeliť danú úlohu na množstvo podúloh, ktoré si privlastňujú zaradom jednotlivé vlákna. To si vyžaduje synchronizáciu pri prístupe do globálneho skladu podúloh.

Často býva značným problémom zabezpečiť efektívne synchronizáciu bez sekventizácie istej časti algoritmu. Spravidla ide o prístupy do dátových štruktúr ako je napríklad zásobník. Rovnako je potrebné zabezpečiť synchronizáciu a výlučný prístup aj v ných častiach algoritmov, čo môže vnášať chyby do implementácií, a tým sa značne zvyšuje ich náročnosť.

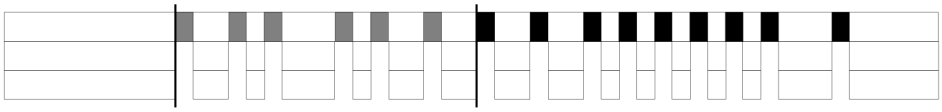
Ďalší problém je v ukončovaní paralelných algoritmov. Použitie paralelizmu vytvára z detekcie ukončenia algoritmu komplexný problém. Jedno jadro môže zisťovať koniec algoritmu, zatiaľ čo druhé práve generuje nové podúlohy. Našťastie existuje niekoľko algoritmov pre zistenie ukončenia práce. Jedno zo správnych riešení je mať jedno jadro, ktoré zisťuje ukončenie algoritmu a mať pre každé jadro atomickú operáciu, ktorou indikuje, či práve pracuje alebo nie. Tu je samozrejme potrebné správne imlementovať protokol pre takúto komunikáciu. Systémy so zdieľanou pamäťou ponúkajú dokonca možnosti, ako všetky vlákna môžu správne zisťovať ukončenie algoritmu.

4.2 Konkurenčné algoritmy

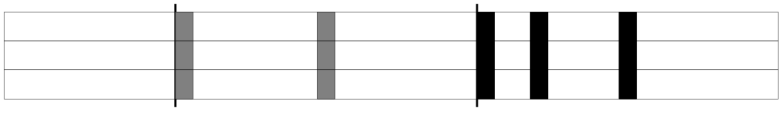
Základné princípy konkurenčného GC boli stanovené pri snahe redukovať dĺžku pozastavenia mutátora pri jednovláknových programoch. Začalo to pri takzvanom inkrementálnom GC. Ten spočíval v prekladaní behu mutátora a GC. Rozdiel od bežného *stop-the-world* je v tom, že cyklus GC bol rozbitý do viacerých pozastavení mutátora. To je vidieť na obrázku 4.1 a). Takýmto rozbitím sa dostávame k niekoľkým problémom. Keďže cyklus GC už nie je atomický vzhľadom k zmene objektov, čiže dosiahnuteľnosť objektov sa môže počas cyklu meniť, inkrementálne GC musia mať spôsob, ktorým tieto možné zmeny dokážu zohľadniť, a tým zabezpečiť bezpečný *garbage collecting*. Napriek tomu, že prekladanie simuluje konkurenčný beh mutátora a GC, k reálnej konkurencii je potrebné urobiť ešte niekoľko krokov. Je potrebné zabezpečiť správnu synchronizáciu tak, aby mutátor aj GC mali konzistentný model haldy. Existujú ešte kompromisy medzi týmito prístupmi, a to je zväčša konkurenčný beh. Ten je popísaný na obrázku 4.1 d). Jeho myšlienka spočíva v tom, že istá časť *garbage collectingu* beží pomocou sto-the-world, kde sa inicializuje cyklus a zvyšok beží konkurenčne s mutátorom. Obrázok 4.1 popisuje jednotlivé prístupy od inkrementálneho k plne konkurenčnému behu mutátora a GC.



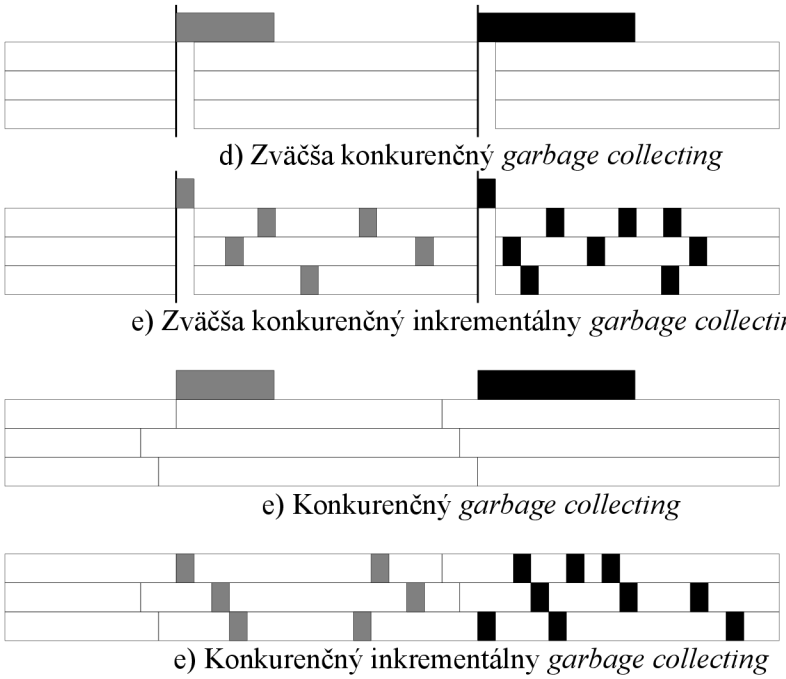
a) Inkrementálny sekvenčný *garbage collecting* pri jednovláknovom mutátore



b) Inkrementálny paralelný *garbage collecting* pri paralelnom mutátore



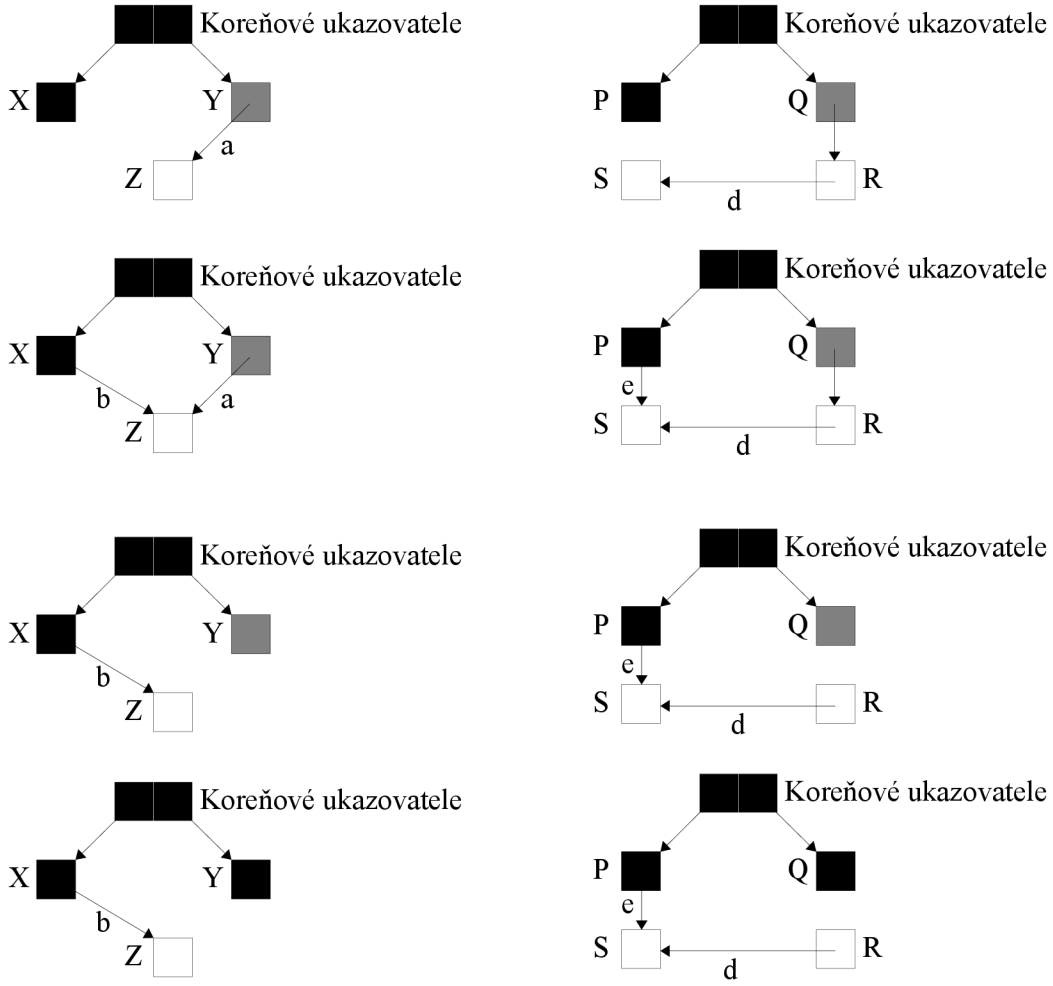
c) Inkrementálny paralelný *garbage collecting*



Obrázok 4.1: Inkrementálny, paralelný a konkurenčný *garbage collecting*. Každý stĺpec reprezentuje jedno vlákno programu na jednom procesore. Rôzne farby častí stĺpcov reprezentujú rôzne cykly *garbage collectingu*. Prebraté z [1].

Pre zabezpečenie korektného *garbage collectingu* je u konkurenčných algoritmov potrebné zachovať bezpečnosť a konečnosť cyklu. Bezpečnosť znamená, že GC nezničí dosiahnuteľné objekty. Konečnosť cyklu je zas požiadavka na GC, aby skôr či neskôr daný cyklus skončil.

V prípade konkurenčného zberu vzniká problém strateného objektu. Na pravej strane obrázka 4.2 sú zobrazené dva scenáre, ako tento problém môže nastať. Prvý scenár nastane, ak sa referencia zo sivého objektu, ktorá ukazuje na biely objekt, presunie do čierneho objektu, ktorý už bol GC spracovaný a zároveň sa daná referencia v sivom objekte zruší. Druhá možnosť nastane, ak mutátor skryje biely objekt, ktorý je tranzitívne dosiahnuteľný cez iný biely objekt. Ilustruje to ľavá strana obrázka 4.2. Je to možné, ak sivý objekt má referenciu na biely objekt X, ktorý má referenciu na iný biely objekt Y. Problém nastane, ak vznikne referencia z čierneho objektu na objekt Y a zároveň sa zruší referencia zo všetkých sivých objektov na objekt X. Podľa [1] problém strateného objektu môže nastať iba ak sú zároveň splnené dve podmienky. Prvá hovorí, že mutátor uloží do čierneho objektu referenciu na biely objekt. Druhá hovorí, že sa zrušia všetky cesty (aj tranzitívne) zo sivých objektov do daného bieleho objektu.



Obrázok 4.2: Problém strateného objektu. Dosiahnuteľný biely objekt je skrytý pred GC, a tým ho GC považuje za nedosiahnuteľný. Vľavo je scenár, kedy je objekt priamo referovaný zo sivého objektu a vpravo scenár tranzitívnej referencie. Prebraté z [1].

Na základe týchto podmienok boli vypracované dva prístupy. Prvý sa nazýva slabá trojfarebná nemennosť referencií, ktorá vraví, že všetky biele objekty, na ktoré smeruje referencia z čiernych objektov, sú sivo kryté, teda existuje aspoň jeden sivý objekt, ktorý má na nich referenciu priamo alebo tranzitívne cez iné biele objekty. Druhá sa nazýva silná trojfarebná nemennosť referencií, ktorá hovorí o tom, že nesmie existovať biely objekt, na ktorý ukazuje čierny objekt. Zatiaľ čo prvý prístup je jednoducho použiteľný v algoritmoch, ktoré nekopírujú objekty, no má problémy s kopírujúcimi, druhý je vhodný pre obe skupiny algoritmov.

Ďalším problémom konkurenčných algoritmov je, kedy je čas začať cyklus. Ak je *garbage collecting* začatý príliš neskoro, môže sa stať, že nezostane dostatok miesta pre alokovanie nových objektov, čo pozastaví beh mutátora, kým sa cyklus neukončí. Pri príliš skorom spustení sa zas môže stať, že nebude dostatok nedosiahnuteľných objektov a zbytočne sa spomalí beh programu.

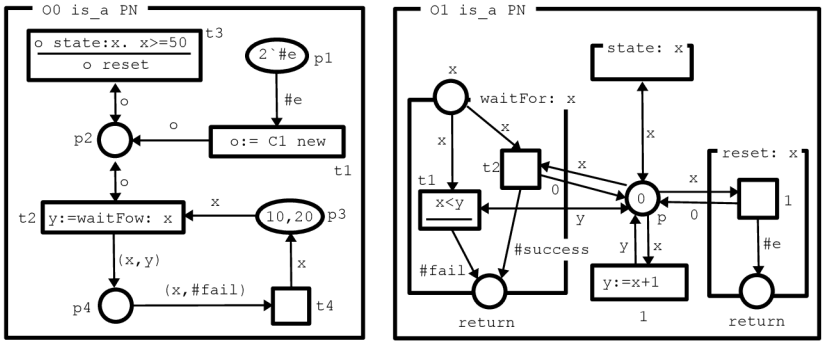
4.2.1 Hodnotenie algoritmov

Konkurenčné *garbage collectory* prichádzajú s novými problémami, ktoré je potrebné riešiť. Sú to hlavne problémy času spustenia a zistenia ukončenia algoritmu, problém strateného objektu, ale aj

problémy týkajúce sa jednotlivých rodín algoritmov. Pre kopírovacie a zhutňovacie algoritmy je to hlavne problém prístupu a zmeny objektov mutátorom, ktoré sú práve kopírované, resp. presúvané. Tieto problémy sú riešené pridaním synchronizačných mechanizmov typu semafor alebo monitor. Tým sa ale spomaľuje čas behu mutátora. Rovnako narastá aj zložitosť samotných algoritmov, čím rastie aj zložitosť implementácie. Ladenie týchto algoritmov je tiež náročnejšie. Napriek vyššie uvedeným problémom sa často tento prístup využíva spolu s generačným prístupom pri spravovaní staršej generácie objektov. Kombinuje sa hlavne konkurenčný paralelný *mark-sweep* s konkurenčným paralelným *mark-compact* algoritmom.

5 Jazyk a systém PNtalk

V tejto kapitole bude popísaný jazyk a systém založený na objektovo orientovaných Petriho sieťach. Jazyk PNtalk je konkrétnou implementáciou objektovo orientovaných Petriho sietí (OOPN). Vychádza priamo z definície (OOPN), má však niektoré syntaktické odlišnosti definované jazykom Smalltalk. PNtalk tiež konkretizuje niektoré skutočnosti, ktoré OOPN ponechávajú voľne definované. Ide hlavne o jednoduché objekty a hierarchiu dedičnosti tried. PNtalk taktiež zavádza niektoré rozšírenia, umožňujúce jednoduchší zápis ako sú zložené správy alebo zoznamy a konštruktory, no ide len o zjednodušenie syntaxe. Jednotlivé prvky jazyka sú demonštrované na obrázku 5.1. Dnes existujú dve implementácie. Prvá je v jazyku Smalltalk [6] a druhá v jazyku C++ [7]. Informácie k tejto kapitole som čerpal z [3] [4] [5][6] a [7].



Obrázok 5.1: Príklad konštrukcie OOPN. Prebraté z [3]

5.1 Motivácia

Petriho siete predstavujú rozšírený formalizmus pre definovanie diskrétnych (paralelných) systémov, ktorý spája výhody zrozumiteľného grafického zápisu a možnosť simulácie s dobrou formálnou analyzovateľnosťou. Praktickej použiteľnosti Petriho sietí v úlohe programovacieho jazyka však bránia významné obmedzenia, ktorými sú statickosť štruktúry siete a jej plošnosť. Inak povedané, Petriho siete neposkytujú štrukturovacie mechanizmy známe z iných programovacích jazykov, ako sú makra, procedúry, funkcie, objekty a podobne. To znamená, že pozitívne vlastnosti Petriho sietí sa prejavia len pri nie príliš rozsiahlych modeloch.

K implementácií rozsiahlych simulačných modelov sa bežne používajú univerzálne programovacie jazyky. Vo vývoji programovacích jazykov je neustála snaha poskytnúť vhodné jazykové konštrukcie pre zjednodušenie tvorby rozsiahlych systémov. Významným medzníkom v tejto snahe bolo vytvorenie objektovej paradigmy. Vzhľadom na značnú komplikovanosť a absenciu formálneho základu sú objektovo orientované modely len veľmi ťažko formálne analyzovateľné. Objektovo orientované jazyky síce formalizujú objektovú orientáciu, ale žiaľ na (implementačne) príliš nízkej úrovni. Rovnako abstraktné objektovo orientované modely, známe z metód objektovo orientovanej analýzy a návrhu programovacích systémov, neumožňujú simuláciu. Je teda opodstatnená snaha vytvoriť na abstraktnejšej úrovni, než je bežný programovací jazyk, vhodnú formalizáciu objektovo orientovaného prístupu a pritom zachovať možnosť simulácie.

Možným riešením je prispôsobenie Petriho sietí požiadavkám, ktoré sú bežne kladené na programovacie jazyky, zavedením vhodnej štrukturalizácie, o čo sa snaží jazyk PNtalk.

5.2 Objektovo orientované Petriho siete

Petriho siete pozostávajú z miest a prechodov, pričom miesto nereprezentuje celkový stav sytému ale iba jeho vlastnosť. Výsledný stav systému je charakterizovaný množinou všetkých ohodnotených miest. Označenie miesta je graficky reprezentované vpísaním značky do miesta. Prechod je následne uskutočniteľný, iba ak systém spĺňa požiadavky pre značenie presetu daného prechodu. Prechod definuje aj postset, čo je množina parciálnych stavov, ktoré systém získa po uskutočnení prechodu. Väzby presetu a postsetu sú graficky znázornené orientovanými hranami od miesta ku prechodu a opačne.

Fakt, že v jednu chvíľu môže byť označených viacero miest poskytuje možnosť modelovať paralelizmus, čo je najväčším prínosom Petriho sietí. Dekompozícia stavu systému na množinu vlastností pozitívne vplýva aj na čitateľnosť modelu, čo je opäť veľkou výhodou. Petriho siete majú viacero rozšírení, no z pohľadu PNtalku sú najpodstatnejšie objektovo orientované Petriho siete.

Definícia objektovo orientovaných Petriho sietí (OOPN) má niekoľko úrovní. Základnú úroveň tvorí definícia systému mien a primitívnych objektov. Tento systém poskytuje primitívnu sémantiku výrazom, ktoré sú súčastou Petriho sietí. Siete sú súčasťou tried. OOPN sú definované systémom tried a špecifikáciou počiatočnej triedy. Okamžitý stav systému je definovaný systémom objektov, opísaných OOPN. Dynamika systému je zas popísaná počiatočným systémom objektov a pravidlami pre generovanie nasledujúcich stavov vykonávaním prechodov Petriho sieťami.

5.2.1 Vlastnosti OOPN

Najdôležitejsím atribútom OOPN je skutočnosť, že samotná Petriho sieť je objekt. Vytvorením objektu nejakej triedy sa automaticky aktivuje jeho objektova sieť, sieť popisujúca hlavnú logiku objektu. Modeluje aktuálny stav daného objektu. Objektová sieť je však skrytá a nedá sa k nej pristupovať zvonku. Okrem nej môže mať trieda metódy a každá metóda má svoju vlastnú Petriho sieť. Tá vzniká zavolaním metódy a zaniká pri skončení metódy. Viacnásobné zavolanie tej istej metódy vytvorí viacero inštancií tejto siete.

Značky sú reprezentované tzv. tokenmi. Tie môžu nadobúdať rôzne hodnoty a môžu niesť aj referencie na objekty, alebo n-tice objektov. Objekt môže byť buď primitívny alebo neprimitívny.

5.2.2 Primitívne a neprimitívne objekty

Nech univerzum zahrňuje množinu primitívnych objektov, množinu mien neprimitívnych objektov a mien tried. Prvky týchto množín sa nazývajú atómy a každému atómu je priradený typ. Každému typu prináleží doména – množina všetkých potenciálnych inštancií typov.

Rozlišujeme primitívne a neprimitívne objekty. Primitívne objektu sú konštanty, ktoré sú implicitne dostupné prostredníctvom svojich mien. Ide spravidla o čísla, textové reťazce, boolovské hodnoty, symboly atď. Keďže ide o konštanty, je možné, na rozdiel od neprimitívnych objektov, primitívne objekty stotožniť s ich menami.

Neprimitívne (definované užívateľom) objekty sú špecifikované triedami a obsahujú stavovú informáciu, ktorá môže byť v priebehu evolúcie systému menená jednak vnútornou aktivitou objektu, alebo vykonávaním metód v dôsledku akceptovania prichádzajúcich správ. Sú špecifikované prostredníctvom Petriho sietí, ktoré sú zložené z miest a prechodov. Miesto môže obsahovať značky, ktoré môžu reprezentovať primitívny objekt, meno triedy alebo n-ticu z nich zloženú. Hrany, ktoré reprezentujú vstupné a výstupné podmienky prechodu, sú označené hranovými výrazmi, ktoré po naviazaní premenných reprezentujú multimnožiny prvkov univerza.

5.2.3 Systém predávania správ a priechody

Je potrebné ešte spomenúť systém správ. Preto je zavedená množina selektorov a špeciálnych správ. Pre primitívne objekty je definovaná sémantika všetkých správ pomocou funkcií. Pre neprimitívne objekty je týmto spôsobom definovaná sémantika len špeciálnych správ, ktoré operujú s menami, nie so stavom, primitívnych objektov (napr.: testovanie rovnosti mien objektov).

Prechod obsahuje stráž a akciu. Stráž obsahuje výrazy (zaslanie správy) a je splnená práve vtedy, ak sú všetky jej výrazy ohodnotené pravdivo. Akcia je zaslanie správy možnosťou priradiť výsledok premennej. Zaslanie správy v akcii prechodu sa interpretuje za behu programu podľa typu adresáta a selektoru správy buď ako primitívne zaslanie, alebo ako vytvorenie nového neprimitívneho objektu, alebo ako žiadosť o vykonanie operácie objektu s čakaním na výsledok (invokácia metódy).

Vzhľadom k potenciálnej možnosti neatomického vykonania prechodu sú zavedené tzv. podmienky prechodu, graficky reprezentované obojstrannými šípkami (testovacie hrany). Ich sémantika je v prípade atomického vykonania prechodu ekvivalentná dvojici vstupnej a výstupnej podmienky (dve opačne orientované hrany) s rovnakým hranovým výrazom. Pri neatomickom vykonaní prechodu, ak je prechodom invokovaná funkcia, vstupná hrana odoberie zo vstupného miesta potrebné značky a výstupná hrana sa uplatní až po vykonaní invokácie. Testovacia hrana však stále ponecháva testované značky v príslušnom mieste.

5.2.4 Triedy a dedičnosť

Triedy môžu byť vytvárané dedením z iných tried, pričom sa pripúšťa len jednoduchá dedičnosť (v súlade so Smalltalk). Špecifikácia triedy obsahuje sieť objektov, siete metód, synchrónne porty a mapovanie správ na metódy a synchrónne porty. Synchrónne porty sú určené k volaniu zo stráži prechodov. Slúži k atomickému testovaniu stavu neprimitívneho objektu v prípade vykonávania prechodu a k jeho prípadnej zmene.

5.3 PNtalk – implementácia v Smalltalk

Behové prostredie Smalltalk už obsahuje vlastný GC, ktorý zbiera a uvoľňuje nedosiahnuteľné objekty. Avšak referencie na objekty PNtalk sú ukladané vo vnútorných štruktúrach tejto nadstavby, konkrétne v slovníkovej kolekcii `components` simulačného sveta `PNtalkWorld` (viac o implementácií simulátoru v kapitole 6.2). Preto GC prostredia Smalltalk považuje tieto objekty počas behu simulácie za dosiahnuteľné a teda ich neuvoľní. Nevplýva to len na permanentné zvyšovanie pamäťových nárokov počas behu simulácie, no značne to znižuje aj výkon PNtalk. Táto kolekcia je často prechádzaná, hlavne pri výbere uskutočniteľného prechodu. Keďže bez uvoľňovania objektov počet objektov v nej v čase rastie, rastie aj čas výberu uskutočniteľného prechodu. To vplýva veľmi nepriaznivo na výkon a použiteľnosť tejto implementácie pre riešenie reálnych simulačných problémov.

V tejto implementácii síce už existuje jednoduchý trasovací GC, ktorý prechádza objektami nedosiahnuteľnými z hlavného objektu a uvoľňuje ich z vyššie spomenutých vnútorných štruktúr, no jeho čas pozastavenia mutátora je príliš vysoký. Pamäť pridelená objektom, ktoré takýto GC uvoľní z vnútorných štruktúr PNtalk je následne recyklovaná GC jazyka Smalltalk. Úlohou diplomovej práce bude v tomto prípade navrhnúť GC, ktorý dokáže takéto nedosiahnuteľné objekty nájsť efektívnejšie. V tomto prípade teda nebude možné zavádzať priamo v GC nejaké optimalizačné metódy, vplývajúce na fyzické uloženie objektov na halde.

5.4 PNtalk – implementácia v C++

C++ implementácia PNtalku je navrhnutá ako generátor C++ kódu modelu z PNtalk kódu modelu a knižnica, implementujúca jednoduchý simulátor. Zo zdrojového textu v PNtalk sa vygenerujú príslušné triedy a simulačný kontext, čiže špecifikácia počiatočnej triedy a volanie spustenia simulácie. Simulácia je následne zostavená skompilovaním modelu a prilinkovaním simulačnej knižnice.

Táto implementácia neobsahuje žiaden GC. Keďže C++ je postavené na explicitnom uvoľňovaní pamäte, nastáva tu teda problém vyčerpávania pamäťových zdrojov pri dlhých simuláciách. C++ síce obsahuje knižnice implementujúce GC, no v tejto práci som sa rozhodol vytvoriť GC zohľadňujúci špecifiká jazyka PNtalk a implementácie simulačnej knižnice. Samotné C++ navyše programátorovi umožňuje absolútnu kontrolu nad uložením objektov v pamäti, čo dáva priestor k optimalizáciám vzhľadom k priestorovej ale aj časovej lokalizácií pomocou GC, čo som sa tiež rozhodol využiť.

6 Návrh spoločnej architektúry automatickej správy pamäte a popis implementácie pre Smalltalk

V tejto kapitole popíšem návrh GC pre jazyk PNtalk. Na začiatku kapitoly je uvedená spoločná architektúra pre obe implementácie a následne sú popísané špecifiká spojené s implementáciou v Smalltalk. V závere uvádzam zhodnotenie tejto implementácie.

6.1 Spoločná architektúra GC

Špecifikum jazyka PNtalk je, že vytvára obrovské množstvo krátko žijúcich objektov. Každá jedna značka je implementovaná ako objekt. Rovnako je tu ale aj druhá skupina objektov, ktoré majú tendenciu žiť dlhšie. Sú to objekty implementujúce siete, miesta, prechody a podobne. Práve pre tieto fakty som zvolil dvojgeneračný GC, kde predpokladám, že krátkožijúce značky budú spravované v mladšej generácií a dlho žijúce objekty budú presunuté do staršej.

Druhý predpoklad, z ktorého som vychádzal, bola tendencia zvyšovať paralelizmus hardvéru. Preto som zvolil spravovanie oboch generácií pomocou paralelných algoritmov, snažiac sa využiť čo najviac možnosti hardvéru. Z vyššie uvedeného dôvodu, že PNtalk vytvára obrovské množtvo objektov, môžem predpokladať, že pri správne nastavených parametroch behu GC bude vždy dostatok práce pre paralelizovanie behu GC.

V kapitole 4.7 uvádzam aj konkurenčné formy algoritmov GC, ktoré sa používajú hlavne pre spravovanie staršej generácie. Tie som ale nepoužil, pretože ich negatíva v prípade implementácií PNtalku prevládajú nad ich pozitívami. Najväčšou devízou konkurenčných algoritmov je absencia pozastavenia behu mutátora. S tým je ale spojená potreba pridania synchronizácie, čo spôsobuje spomalenie mutátora pri prístupe k objektom a hlavne ladenie a vývoj virtuálneho stroja robí zložitejším. Obidva problémy sú v tomto prípade kritické, pretože mutátor v oboch implementáciách má výkonnostný problém pri reálnych simuláciách a keďže obe implementácie vznikli na akademickej pôde, predpokladá sa ich vývoj mnohými programátormi a rovnako sa predpokladajú ešte zavedenia mnohých optimalizácií, ktoré by neboli po implementovaní konkurenčných algoritmov možné.

Podľa kapitoly 4.1 je potrebné si odpovedať ešte na dve ďalšie otázky. Prvou z nich je vyvažovanie záťaže medzi vláknami. Keďže statické rozdelenie by nevyužívalo efektívne paralelizmus hardvéru, niektoré vlákna by mali množstvo práce a iné by ostávali nevyužité, zvolil som dynamické vyvažovanie. Keďže v smalltalku nie je možnosť priamo pracovať s umiestnením objektov na halde, zvolil som procesocentrický prístup. Tento prístup som využil aj v implementácii v C++, kde bola výhoda procesocentrického prístupu lepšie rozdelenie záťaže. Pri pamäťocentrickom prístupe by bolo potrebné si udržovať pre každú časť pamäte buď počet objektov v nej, alebo jej počiatok, čo by malo nepriaznivý vplyv na rýchlosť a pamäťové nároky behu mutátora.

6.1.1 Procesocentrické vyvažovanie záťaže a problém zastavenia

Efektívny spôsob implementácie tohto prístupu, je každé vlákno vybaviť privátnym zoznamom s položkami pre spracovanie. Ako kolekciu som zvolil frontu. Do tejto fronty vlákno vkladá sivé

objekty a následne vyberá objekty, ktoré aktuálne spracúva (mení na čierne). Ak vlákno vyberie všetky prvky fronty, začne prechádzať cyklicky ostatné vlákna, až kým nenarazí na vlákno s neprázdnou frontou. Následne z tejto fronty vyberie najviac polovicu jej položiek a vloží ich do vlastnej. To popisuje algoritmus 6.1 v sekcii 2. Tu je ale potrebné pridať synchronizáciu pri vkladaní aj výbere z fronty, pretože viacero vláken môže do nej pristupovať. Je potrebné ešte spomenúť, prečo najviac polovicu prvkov. Zatiaľ čo vlákno *a*, ktoré odoberá pre seba prvky z fronty vlákna *b*, tak vlákno *b* tiež z nej odoberá prvky, ktoré aktuálne spracúva. Podobne, viacero vláken môže naraz odoberať prvky z fronty jedného vlákna (diagram 6.1). Ak teda vlákno *b* stihne odobrať viac než polku svojej fronty, pre vlákno *a* zostane menej ako polovica, ktorú zoberie do vlastnej fronty.

S týmto postupom súvisí aj problém zastavenia, teda, ako všetky vlákna správne zistia, že je koniec algoritmu. Podľa [1] existuje viacero prístupov. Je možné zaviesť globálne počítadlo prvkov, no jeho aktualizácia viacerými vláknami znižuje mieru paralelizmu a zavádza silne sekvenčné miesto. Upravil som teda algoritmus uvedený v [1], ktorý predpokladá dva indikátory indikujúce ukončenie práce vlákna. Každé vlákno, ktoré si vyprázdni svoju frontu a prejde dvakrát cyklicky ostatné vlákna, pričom každé z nich má v daný čas prázdnu frontu, tak sa ukončí. Takéto prechádzanie a testovanie plnosti front je dostatočne rýchle na to, aby vlákno, ktoré má dostatočne plnú frontu z nej nestihlo vyprázdniť všetky objekty. Aj keď existujú prípady, kedy tento algoritmus falošne detekuje ukončenie niektorými vláknami, je dostatočne rýchly a takéto prípady sa vyskytujú veľmi zriedka, čo som si overil pri testovaní, kde nevznikol ani jeden takýto problém. Celý algoritmus je uvedený ako algoritmus 6.1.

```
threadWork(unsigned threadIdx):
    unsigned threadsCnt = threads.size()
    while (true):
        /* Sekcia 1: spracovanie položiek */
        while !privateQ.empty():
            T item = privateQ.popFront()
            queue newItems = process(item)
            privateQ.append(newItems)

        /* Sekcia 2: kontrolovanie ostatných vláken a preberanie
         * položiek z ich privátnych front */
        for (unsigned i = 0; i < 2*threadsCnt; i++):
            queue tmpQ = threads[i%threadsCnt].stealHalfQ()
            if (!tmpQ.empty()):
                privateQ.append(tmpQ)
                break

        /* Section 3: if no work aquired, thread is going to die */
        if (privateQ.empty()):
            break
```

Algoritmus 6.1: Implementácia procesocentrického vyvažovania záťaže s ukončujúcou podmienkou.

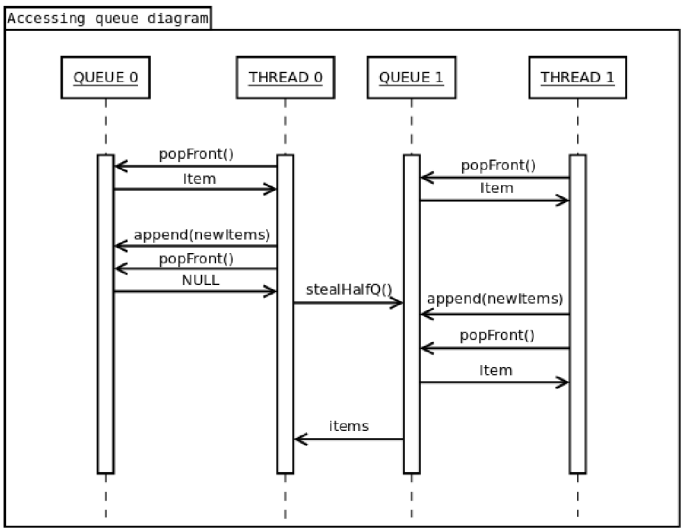
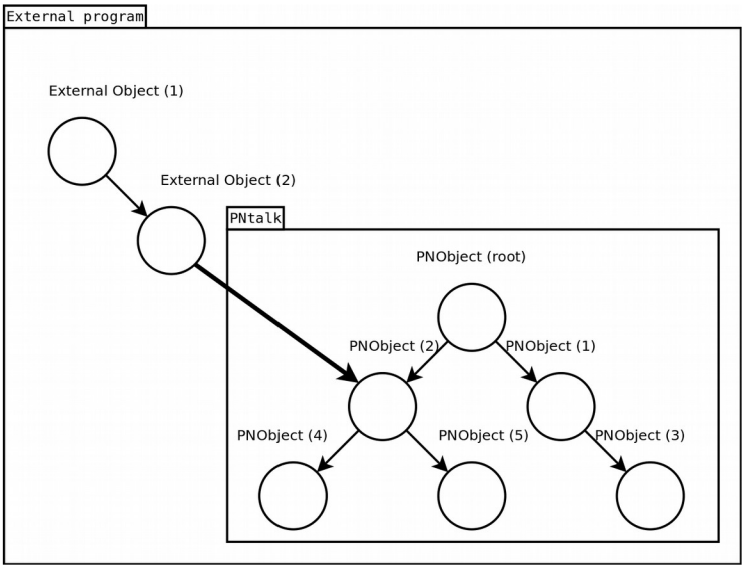


Diagram 6.1: Diagram prístupu vláken k frontám s prípadom, kedy viacero vláken pristupuje k privátnej fronte vlákna 1.

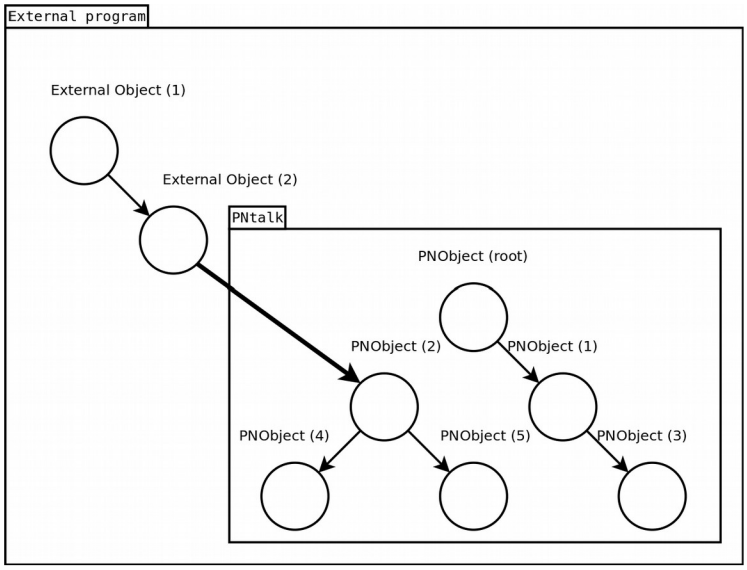
6.1.2 Problém externých referencií

Externými referenciami chápeme referencie na PNtalk objekty z objektov mimo prostredia PNtalk. Je potrebné zaručiť, aby GC takéto objekty neuvoľnil predčasne, teda v situácií, kedy na PN objekt neexistuje žiadna referencia z prostredia PNtalk, no existuje aspoň jedna referencia z objektu mimo prostredia PNtalk. Tento prípad ilustruje obrázok 6.1. Uvažovanie takýchto referencií umožňuje použiť PNtalk ako súčasť väčšieho simulačného celku, čiže simulovať len určité časti modelu pomocou PNtalk a iné časti pomocou iných techník, ktoré umožňuje prostredie, v ktorom je daná implementácia PNtalku vytvorená. Spravovanie takýchto externých referencií musí rešpektovať správne praktiky programovania v onom použitom prostredí, hlavne čo sa týka automatickej, resp. explicitnej správy pamäte.

a)



b)



Obrázok 6.1: Problém externých referencií. Po zrušení referencie z PNObject(root) na PNObject(2) nesmú byť objekty PNObject(2), PNObject(3) a PNObject(4) uvoľnené, pretože na nich ukazuje externý obejkt ExternalObject(2). Externá referencia je znázornená hrubšou šípkou.

6.2 Smalltalk implementácia

Ako som už spomínal v kapitole 5.3, pri implementácií v Smalltalku sa problém redukuje na označenie živých objektov a na efektívne prechádzanie zoznamom objektov. Presne tieto dve činnosti robí *mark-sweep* algoritmus, ktorý som zvolil pre spravovanie oboch generácií. Na prvý pohľad sa môže zdať dvojgeneračný prístup menej efektívny, ako jednogeneračný, keďže sa graf objektov musí celý prechádzať aj pri cykle staršej aj pri cykle mladšej generácie, no jeho efektivita spočíva v následnej *sweep* fáze, kde sa musí prechádzať len menej objektov a tým sa znižuje čas pozastavenia. Po ďalšie počítame, že pri behu simulácie vzniká množstvo objektov, ktoré žijú krátko a len niektoré žijú dlhšie. Pri prechádzaní grafu objektov sa teda prejde omnoho menej objektami ako pri prechádzaní zoznamu objektov, teda prechádzanie grafom objektov je omnoho rýchlejšie ako prechádzanie zoznamu objektov.

Problémom v Smalltalku je, že nepozná ukazovatele do dátových štruktúr a teda povoľuje vytvárať podzoznamy iba za pomoci kopírovania a vytvárania nových zoznamov, čo je pomalé. Je teda potrebné už pri vytváraní objektov a registrovaní v zozname objektov myslieť na paralelné prechádzanie a uvoľňovanie objektov.

6.2.1 Popis existujúcej implementácie

Implementácia v Smalltalk obsahuje niekoľko kľúčových tried, ktoré je potrebné k fungovaniu GC spomenúť. Prvou z nich je trieda `PNtalkWorld`, ktorá si drží zoznam `components` všetkých objektov vytvorených v PNtalk. Práve tento zoznam bráni GC prostredia Smalltalk zozbierať nedosiahnuteľné objekty. Táto trieda ďalej implementuje metódy vykonávajúce predávanie správ medzi objektami a volanie jednotlivých metód zo správ. `PNtalkWorld` obsahuje aj kalendár udalostí a implementuje metódy pre prácu s ním.

Ďalšou dôležitou triedou je `PNtalkSimulation`. Tá riadi beh simulácie. Pomocou nej sa dá simulácia spustiť ale aj pozastaviť resp. zastaviť. Rovnako implementuje aj možnosť zistiť stav simulácie, či simulácia beží, alebo je pozastavená.

Objekty jazyka PNtalk dedia od spoločnej triedy `PNObject`, ktorá implementuje nutné spoločné správanie všetkých objektov. Táto trieda si drží informáciu o jedinečnom ID objektu a implementuje metódy pre vrátenie všetkých referencií, ktoré daný objekt obsahuje, resp. referencií na daný objekt.

Referencia na objekt je implementovaná ako objekt triedy `PNtalkProxy`. Ak chce získať nejaký objekt referenciu na objekt PNtalku, vytvorí sa nová instancia `PNtalkProxy`, cez ktorú bude možné pristupovať k danému objektu. Takéto správanie zabezpečuje, že objekt nie len že vie o referenciách, ktoré má na iné objekty, ale vie aj o referenciách, ktoré ukazujú z iných objektov naň. Zabezpečuje to aj možnosť odlíšiť externú od internej referencie, čo je popísané nižšie.

Poslednou triedou, ktorú je potrebné spomenúť, je trieda `PNCompiledClass`. Táto trieda implementuje triedy PNtalk. Obsahuje atribúty ako rodičovská trieda, zoznam objektov tejto triedy, zoznam portov, alebo metódy pre vytvorenie a registrovanie objektu v simulácií. Práve tieto metódy sú využité pri riešení problému externých referencií (popis v kapitole 6.2.4).

6.2.2 Úpravy a rozšírenia existujúcej implementácie

Vlastná implementácia vyššie uvedených myšlienok spočívala v rozšírení niekorých existujúcich tried a v zavedení nových tried GC. Problém paralelného prechádzania zoznamu objektov vo sweep fáze je riešený už pri vytváraní nových objektov. Trieda `PNtalkWorld` bola teda rozšírená o zoznamy pre staršiu a mladšiu generáciu objektov a o cyklické čítače ukazujúce na tieto zoznamy. Tie určujú, kam sa má registrovať novovzniknutý, objekt, resp. objekt presunutý do staršej generácie. Nové objekty sa teda striedavo ukladajú do *n* zoznamov reprezentujúcich jednu generáciu, pričom *n* je počet vláken GC. To zaručí, že každý zoznam má približne rovnako veľa objektov. Následne vo sweep fáze každé vlákno prechádza jedným takýmto zoznamom. `PNtalkWorld` je ďalej rozšírená o metódy pre presun objektov z mladšej do staršej generácie (`moveToSecondGeneration`) a uvoľňovanie objektov z jednotlivých generácií (`removeComponentNamed: aName generation: gen`).

Ďalej bola upravená trieda `PNObject`, kde sa pridal atribút značky (`marked`), semafor pre synchronizáciu označovania (`gcSem`) a počítadlo pre aktuálny počet prežitých cyklov, na základe ktorého sa presúva do staršej generácie (`gcCounter`). Rovnako v ňom boli implementované metódy pre nastavovanie a testovanie značky (`mark: aMark`, `marked`, `testAndSetMark: aMark`), prácu so semaforom (`gcSemWait`, `gcSemSignal`, `gcSemIsSignaled`), metódy na inkrementáciu a testovanie počítadla (`gcCounted`, `addGCCounter: aNum`) a metóda vracajúca všetky referencie, ktoré daný objekt má v sebe uložené (`allReferences`) na iné objekty.

Hlavnými triedami GC sú `PNtalkGarbageCollectorNew`, `PNtalkGCMarkingThread` a `PntalkGCSweepingThread`. Prvá z vyššie spomínaných zaobaľuje GC a implementuje plánovanie a iniciovanie cyklov. Rovnako sa v nej dajú nastaviť parametre GC (počet vláken, počet cyklov, po ktorých je objekt presunutý do staršej generácie, perióda cyklov staršej / mladšej generácie) a to v metóde `initialize`. `PNtalkGCMarkingThread` implementuje marking fázu pre jednotlivé vlákna. Okrem toho sa v nej nachádza privátna fronta (`privateQueue`) a metódy pracujúce s ňou (`getHalfQ`). Posledná z

vyššie uvedených tried implementuje paralelné prechádzanie zoznamom objektov, ich uvoľňovanie a rušenie nastavenej značky.

6.2.3 Popis behu GC

Beh cyklu GC sa deje na konci simulačného kroku, a to vždy po presne stanovenom počte takýchto simulačných krokov. Na začiatku cyklu GC sa rozbieha marking fáza na jednom vlákne (algoritmus 6.2), pričom pri každom prechode na nový objekt sa kontroluje veľkosť privátnej fronty. Ak veľkosť privátnej fronty prekročí počet vláken, fronta sa rozdelí rovnomerne do privátnych front pre každé vlákno a začína sa paralelná marking fáza (algoritmus 6.3, prvá časť).

Po ukončení cyklu (algoritmus 6.3 druhá časť) sa spúšťa paralelná sweep fáza, kde jednotlivé prvky sú buď uvoľnené, presunuté do staršej generácie, alebo je im inkrementované počítadlo prežitých cyklov. Tu je potrebné podotknúť, že pri jednotlivých cykloch sa musí meniť značka. Pre označovanie objektov pri zbere mladšej generácie som určil za značky čísla 1 alebo 2, pričom novovzniknutý objekt má značku 0. Pre zber staršej generácie som určil za značky čísla 3 a 4. Toto riešenie je potrebné, keďže pri zbere staršej generácie sú síce označené aj všetky objekty, ktoré patria do mladšej generácie, no následne pri *sweep* fáze sú zrušené značky len starších objektov. Objekty patriace do mladšej generácie ostávajú teda označené a teda pri následnom cykle mladšej generácie by neboli uvoľnené aj napriek tomu, že neboli v mark fáze označené. To isté platí aj pre objekty staršej generácie pri cykle GC pre mladšiu generáciu. Tento problém ilustruje obrázok 6.1. Cykly GC sú ukončené, keď sa vyprázdni fronta správ. Pri jej opätovnom naplnení sa GC opäť spúšťa.

```
void threadMark(queue privateQ, unsigned mark):
    unsigned threadsCnt = THREADS.size()
    while (not privateQ.isEmpty() and privateQ.size()< threadsCnt):
        PNObject actObject = privateQ.popFront()
        // musí byť synchronizované, kvôli datarace
        bool marked = (actObject.testAndSetMark() == mark)
        if (not marked):
            PNObject* references = actObject.getReferences()
            privateQ.pushAll(references)

/* Po návrate je potrebné ešte raz kontrolovať veľkosť fronty,
   pretože táto metóda môže skončiť aj tým, že prešla všetky živé
   objekty. Potom privateQ.size() == 0 */
```

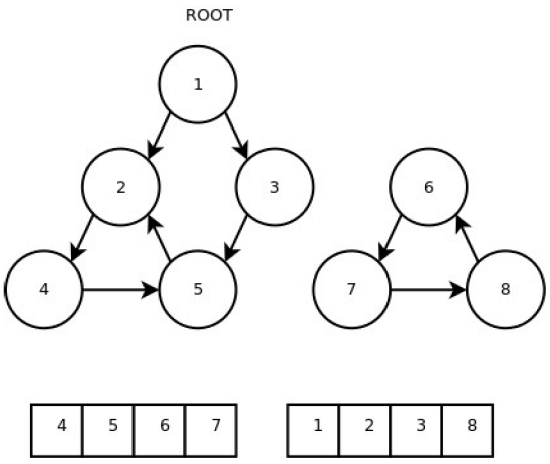
Algoritmus 6.2: Štart marking fázy na jednom vlákne, ktoré následne končí, keď je dostatok položiek v privátnej fronte pre paralelizovanie.

```
void collect():
    unsigned threadsCnt = THREADS.size()
    queue privateQ
    /* Potrebné, kvôli dvom generáciám, pri prechádzaní jednej
       generácie sa neodstránia značky z druhej, preto sa musia
    značky striedať */
    unsigned mark = getNewMark()
    PNObject* roots = getRoots()
    privateQ.pushAll(roots)
    threadMark(privateQ, mark)
    /* Je potrebné ešte raz kontrolovať veľkosť fronty,
```

```
   pretože threadMark metóda môže skončiť aj tým, že prešla
   všetky živé objekty. Potom privateQ.size() == 0 */
if (not privateQ.isEmpty):
    /* Prvá časť, spustenie paralelnej marking fázy */
    MThread markingThreads[threadsCnt] =
      createMarkingThreads(threadsCnt, mark)
    /* rovnomerné rozdelenie privateQ medzi privátne fronty
       jednotlivých vláken */
    partitiateEqualyQintoPrivateQs(privateQ, markingThreads)
    foreach thread in markingThreads:
        thread.start()
    foreach thread in markingThreads:
        thread.waitForFinish()
/* Druhá časť, spustenie paralelnej sweeping fázy */
SThread sweepingThreads[threadsCnt] =
  createSweepingThreads(threadsCnt, mark)
for (unsigned i = 0; i < threadsCnt; i++):
    sweepingThreads[i].setList(OBJECT_LISTS[i])
    sweepingThreads[i].start()
foreach thread in sweepingThreads:
    thread.waitForFinish()
/* Už sú uvoľnené všetky vnútorné referencie nedosiahnuteľných,
   objektov, je teda potrebné spustiť vstavaný GC, ktorý uvoľní
   pamäť */
SystemGC.execute()
```

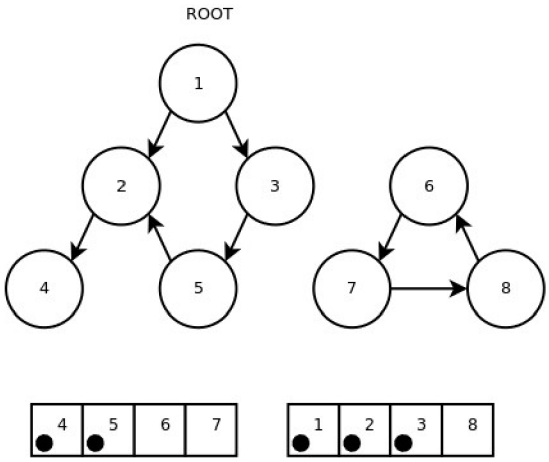
Algoritmus 6.3: Algoritmus zberu jednej generácie.

a)



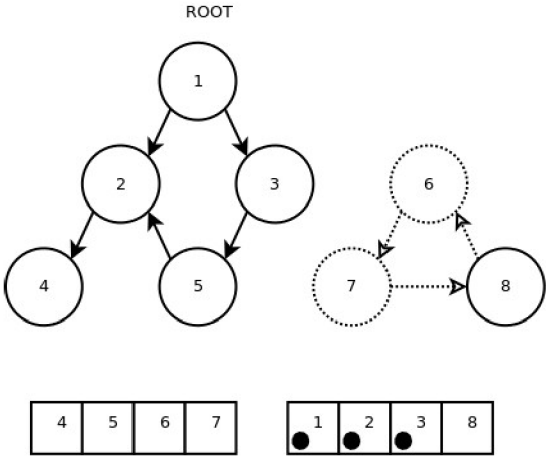
Stav pred prvou marking fázou. V hornej časti je graf objektov s naznačeným rootom a v dolnej ľavej zoznam objektov mladšej generácie a vpravo staršej.

b)



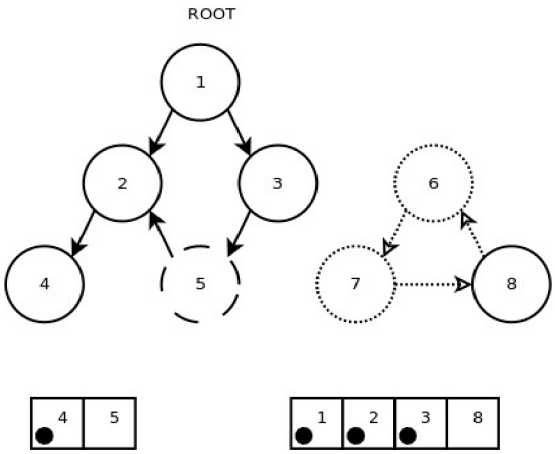
Stav po prvej marking fáze. Bodka v zoznamoch reprezentuje označený objekt.

c)



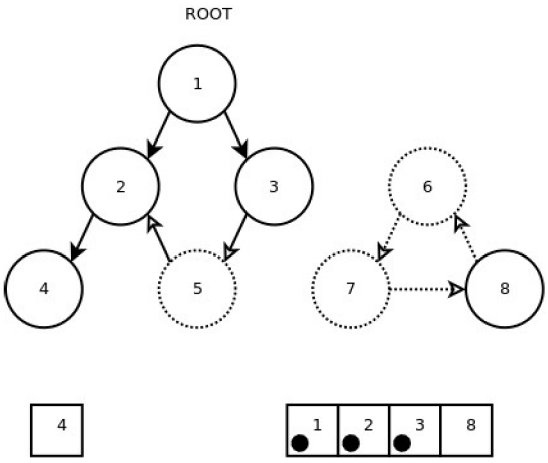
Stav po sweep fáze mladšej generácie. Čiarkované objekty v grafe boli uvoľnené. Ako je vidieť, v zozname staršej generácie neboli značky vymazané.

d)



Stav po ďalšej marking fáze. Čiarkovaný objekt 5 nebol označený, aj keď je dosiahnuteľný, pretože objekt 3 ostal v staršej generácii označený z predošlej marking fázy a teda nebol v tejto marking fáze spracovaný.

e)



Stav po sweep fáze mladšej generácie. Objekt 5 bol predčasne uvoľnený, keďže nebol označený.

Obrázok 6.1: Názorná ukážka problému predčasného uvoľnenia objektu, ak nie sú značky pri jednotlivých cykloch GC striedané.

6.2.4 Externé referencie

Problém externých referencií v tejto implementácii vzniká, ak je v prostredí Smalltalk vytvorený PNtalk objekt, ktorý je následne registrovaný v PNtalk svete a tým sa stáva súčasťou simulácie, no ukazuje naň ešte referencia zo Smalltalk. Ako som uviedol v kapitole 6.2.1, referencie v PNtalku sú

modelované pomocou tzv. proxy objektov. Objekt PNtalk vie o všetkých proxy objektoch, ktoré naň ukazujú. Pri vzniku referencie vznikne proxy objekt, ktorý ju reprezentuje a pri jej zániku takýto objekt zaniká.

Riešenie problematiky externých referencií spočíva teda vo vytvorení atribútu v proxy objekte, na základe ktorého sa dá rozlíšiť interný proxy objekt (medzi dvoma objektami PNtalku) a externý proxy objekt (referencia z externého objektu na objekt PNtalk). Bolo potrebné rozšíriť triedu reprezentujúcu objekty PNtalk o atribút `hadProxy`, ktorý je nastavený pri vytvorení externej referencie na daný objekt a metódu `hasProxy`, ktorá iteruje cez všetky referencie na daný objekt a zistí, či aktuálne naň existuje externá referencia. Atribút `hadProxy` slúži pre performačné účely, aby sa pre každý objekt nemusela volať metóda `hasProxy`, ktorá musí iterovať cez všetky referencie na objekt, ale aby sa táto metóda volala len pre objekty, pre ktoré to má zmysel.

Pri vytvorení externej referencie na objekt sa nastaví v objekte atribút `hadProxy` a objekt sa pridá do zoznamu root objektov, teda objektov považovaných za vstupné body programu. Práve od týchto objektov začína trasovanie GC. Marking fáza ostáva bez zmeny, no mierne bolo potrebné upraviť *sweep* fázu. V tej sa pri prechádzaní objektov na začiatku spracovávania objektu testuje atribút hadExternal. Ak objekt má nastavený tento atribút na true, zistí sa, či objekt má ešte stále nejakú externú referenciu pomocou metódy `hasExternal`. Ak objekt takúto referenciu už nemá, odstráni sa zo zoznamu *root* objektov a nastaví sa `hadExternal` na false. Zvyšok *sweep* fázy ostáva rovnaký. Celá úprava *sweep* fázy je popísaná v algoritme 6.4.

Pre zjednodušenie vytvárania externých objektov a ich registrácie do simulácie bola vytvorená špeciálna metóda triedy `PNCompiledClass createIn`. Táto metóda vytvorí objekt danej triedy, nastaví atribút `hadExternal` na true a tento objekt následne registruje v simulácií. Metóda vráti proxy na vytvorený objekt s nastavením atribútu, že sa jedná o externú proxy.

```
void sweep(PNtalkWorld world, list listOfObjectsInGeneration):
    unsigned idx = 0;
    while (idx < listOfObjectsInGeneration.size()):
        PNObject actObject =  listOfObjectsInGeneration[idx]

        /* Vysporiadanie sa s externými objektami */
        if (actObject.hadExternal()):
            if (not actObject.hasExternal()):
                actObject.setHadExternal(false)
                world.removeFromRoots(actObject)

        /* zvyšok sweep fázy */
        if (actObject.isMarked()):
            actObject.removeMark()
            /* zvýši počet prežitých cyklov GC */
            actObject.increaseGCCyclesCnt()
            /* isOldEnought vráti true, ak objekt prežil dostatočne
               veľa cyklov GC v mladšej generácií */
            if (actObject.isOldEnought() and sweepingYoung()):
                /* predpokladám, že sa zníži počet objektov v
                   listOfObjectsInGeneration, teda nezvyšujem idx */
                world.moveToOldGeneration(actObject)
            else:
                idx++;
        else:
```

```
/* objekt sa musí odstrániť zo zoznamu
   listOfObjectsInGeneration aj zo zoznamu všetkých
   objektov PNtalk simulácie */
world.removeObject(actObject)
```

Algoritmus 6.4: Algoritmus implementujúci sweep fázu s uvažovaním externých referencií.

6.2.5 Zhodnotenie riešenia a možné optimalizácie

Implementácia využíva vo veľkej miere paralelný hardvér, čo zvýšilo výkon GC. Miestom, ktoré beh algoritmu z časti sekventizuje je prístup do privátnych front jednotlivých vláken. Je možné zaviesť synchronizáciu len pri prístupe ku poslednému prvku, no bolo by potrebné implementovať vlastnú kolekciu, keďže pri terajšej implementácií použitých kolekcií by takéto zníženie synchronizácie zaviedlo chybné nastavovanie atribútov fronty, ako je napríklad jej veľkosť.

Problém tejto implementácie je, že implementácia PNtalk neposkytuje informáciu o ukončení simulácie a testovanie na prázdnosť fronty správ detekuje aj prípady, kedy simulácia ešte nie je ukončená. Riešenie tohto problému si žiada hĺbkovú zmenu fungovania simulačného cyklu, čo by bolo mimo rozsah tejto práce.

Hoci celkový výkon aplikácie vzrástol, pridanie atribútov do objektov, potrebných pre beh GC, zvýšilo pamäťové nároky. To predstavuje priestor pre ďalšiu optimalizáciu a vývoj. Je možné napríklad zaviesť miesto oddeleného atribútu pre počet prežitých cykov GC a značku, ktoré nevyužívajú všetky možné hodnoty atribútov, jediný atribút, ktorého horné tri bity budu určovať značku a spodné cyklus GC. Ďalšou možnosťou je zaviesť pre značky bitovú tabuľku, no to by mohlo nepriaznivo ovplyvniť priestorovú lokalitu, keďže by sa pri prechádzaní objektu muselo skákať stále do tabuľky, ktorá by bola umiestnená v pamäti inde, ako daný objekt.

7 Popis implementácie automatickej správy pamäte pre C++ PNtalk

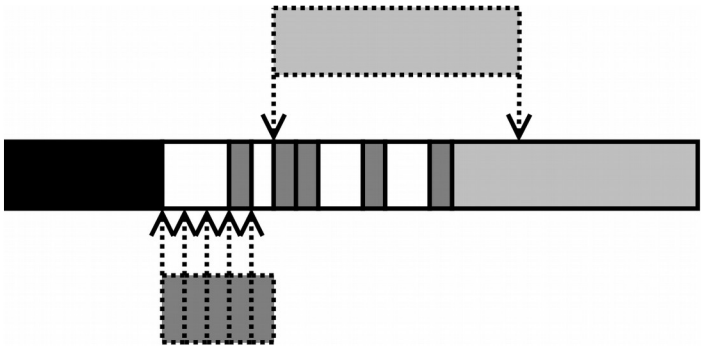
Ako bolo uvedené vyššie, implementácia PNtalk v C++ pozostáva z dvoch programových častí. Prvá je generátor kódu modelu z PNtalk do C++ a druhá je knižnica, ktorá implementuje beh simulácie pre daný model. C++ programátorovi poskytuje množstvo slobody pri práci s pamäťou a pri ukladaní objektov v nej. Preto mojím cieľom bolo zaviesť čo najviac pamäťových optimalizácií pre beh mutátora a tým kompenzovať čas stratený pri behu GC.

7.1 Architektúra GC a použité algoritmy

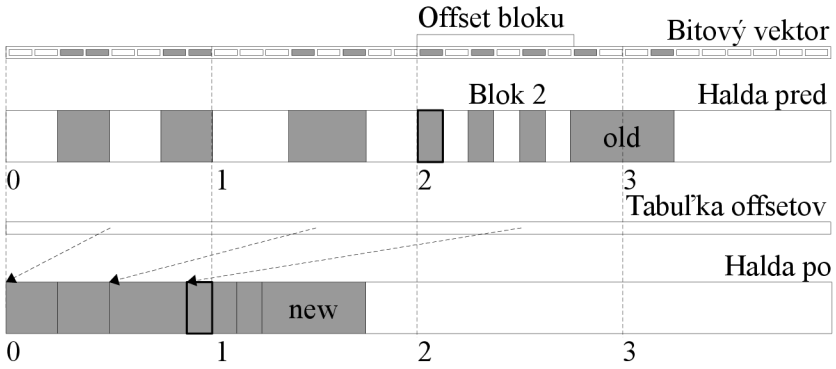
Pre správu mladšej generácie som použil paralelný kopírovací algoritmus. Ten podľa [1] zlepšuje priestorovú lokalitu objektov, zabraňuje fragmentácii pamäte a zrýchľuje alokáciu objektov, čo priaznivo vplýva na výkon mutátora.

Pre správu staršej generácie som tiež použil paralelný kopírovací algoritmus. Táto voľba bola ovplyvnená tým, že implementácia PNtalk v C++ generuje množstvo malých objektov, z ktorých veľká časť sa dostane do staršej generácie. Jedná sa hlavne o Tokeny, ktoré prežívajú množstvo prechodov. Použitím paralelného *mark-compact* by teda potrebná synchronizácia tvorila značné spomalenie a serializovanie algoritmu. Jednalo by sa hlavne o prípady, kedy by bol presúvaný veľký objekt na miesto, kde sa ešte nachádza množstvo menších nepresunutých. Situáciu ilustruje obrázok 7.1. *Mark-compact* si rovnako vyžaduje dvojité prechádzanie haldou, čo opäť znižuje výkon GC. Výhoda mark-compact je ale v menšej pamäťovej náročnosti. Teoretická pamäťová náročnosť je 50% pamäte, ktorá je potrebná pre kopírovacie algoritmy. Treba však započítať ešte miesto pre uloženie počiatočnej adresy každého objektu. To je riešené v paralelnom *mark-compact* algoritme zostavením bitového vektora a tabuľky offsetov, kde každý bit prislúcha jednému bajtu alokovanej pamäte. Daný bitový vektor je nastavený, ak na odpovedajúcom byte sa nejaký objekt začína, alebo končí (ilustrácia obrázok 7.2). Tým sa pridá minimálne 1/8 veľkosti alokovanej haldy k pamäťovej náročnosti. Na druhej strane takýto bitový vektor rapídne zníži vhodnosť priestorovej lokality, keďže pri prechádzaní objektov sa musí často pracovať s týmto vektorom, ktorý môže byť uložený v inej časti pamäte, ďaleko od práve spracovávaného objektu.

Konkurenčný algoritmus som opäť nepoužil z dôvodov uvedených v kapitole 6.1, čo je hlavne zneprehľadnenie kódu, potreba ďalšej synchronizácie pri prístupe k objektom, na ktorú sa pri použití kopírovacích algoritmov kladú ešte väčšie nároky a tým je zložitejšia, čo spomaľuje beh mutátora a zmenšenie priestoru pre iné optimalizácie a zmeny v implementácii.



Obrázok 7.1: Názorná ukážka problému mark-compact algoritmu. Algoritmus musí najprv presunúť množstvo tmavosivých objektov, pred tým než presunie veľký svetlosivý. Vlákno, ktoré chce presúvať svetlosivý musí teda čakať.



Obrázok 7.2: V hornej časti obrázku je bitový vektor s každým bitom určeným pre jeden bajt haldy. Bit je nastavený tam, kde sa dosiahnuteľný objekt začína a končí. To umožňuje počítať adresu presunutia objektu pri paralelnom mark-compact algoritme za pomoci ďalšej prídavnej tabuľky offsetov. Prebraté z [1].

7.2 Správa pamäte a optimalizácie

Ako som uviedol vyššie, snažil som sa čo najviac kompenzovať beh GC zrýchlením behu mutátora. Prvú optimalizáciu, ktorú som zaviedol, bolo urýchlenie alokácie a zrušenie volania deštruktorov. Myšlienka bola získať na začiatku dostatočne veľkú časť pamäte od OS, a následne v nej alokovať nové objekty. Alokácia pamäte operačným systémom je časovo drahá operácia, preto pri častom volaní (alokácií nových objektov) môže nepriaznivo ovplyvňovať beh mutátora. Alokácia nového objektu je teda redukovaná len na posuv ukazateľa na prvú voľnú adresu o veľkosť objektu. Rovnako odstránenie volania deštruktorov, ktoré pri kopírovacích algoritmoch nie je priamo možné, a zrušenie častých volaní uvoľňovania pamäte prispievajú tiež k zrýchleniu behu mutátora. *Fromspace* po migrovaní všetkých živých objektov môže byť naraz vrátené operačnému systému, čím sa ušetrí množstvo volaní *free*. V prípade PNtalk sú tieto optimalizácie ešte viac výhodné, keďže pri behu vytvára a ruší veľké množstvo objektov. Okrem týchto výhod takáto sekvenčná alokácia objektov priaznivo vplýva na priestorovú lokalitu objektov, čo má opäť vvplyv na urýchlenie behu mutátora.

Jediným negatívom je, že po behu GC je potrebné aktualizovať obsah všetkých vyrovnávacích pamätí.

Pre zavedenie týchto optimalizácií bolo potrebné vytvoriť vlastné kolekcie. V implementácii sú použité kolekcie *list* a *vector*. Implementácia vlastného alokátora a použitie už existujúcich implementácií by nebola vhodná, pretože funkčnosť štandardných kolekcií je potrebné rozšíriť o množnosť potrebnú pre beh GC. Jedná sa hlavne o získanie referencií na všetky objekty, registrovanie typu objektov, ktoré sa v danej kolekcii nachádzajú, a to objekt PNtalku, ukazovateľ na objekt v PNtalku a iný objekt. V prvom prípade sa musia pri volaní `getReferences` vrátiť všetky referencie každého objektu v kolekcii. V druhom prípade stačí vrátiť referencie na každý objekt v kolekcii a v poslednom prípade nie je potrebné vrátiť nič. Pri zhodnotení absencie virtuálnych deštruktorov a možnosti plne prispôsobiť implementáciu kolekcií potrebám GC a mutátora som zvolil vlastnú implementáciu vyššie uvedených dvoch kolekcií, ktorá je funkčne ekvivalentná referenčnej implementácii zo štandardnej knižnice v rozsahu použitom v programe vrátane iterátorov. Následne bolo potrebné nahradiť volanie všetkých operátorov `new` operátorom `new(void *ptr)`, kde `ptr` je ukazovateľ na už alokované miesto. Pre získanie ptr som vytvoril makro `GC_ALLOC(GC, OBJECT)` volajúce alokáciu miesta pre uloženie `OBJECT` riadenú GC.

Je potrebné spomenúť, že kvôli implementácio týchto vlastných dátových štruktúr, je interpret obmedzený na približne 2 000 000 objektov spravovaných pomocou GC (max. 7 zanorení v príklade z prílohy B1). Možnosť akceptovať viac objektov by spočívala vo zvýšení pamäťovej náročnosti, čo na základe výsledkov z kapitoly 8 neprichádza do úvahy bez zavedenia optimalizácií, ktoré sú popísané v 7.6.

Zaujímavosťou je efektívnosť implementácie týchto vlastných kolekcií. V prvej verzii som implementoval iba kolekciu vector, ktorú som následne používal aj na miestach, kde bol pôvodne list. Takáto implementácia bola ale príliš pomalá. Doimplementovaním kolekcie *list* a zavedením efektívnej implementácie iterátorov časová náročnosť implementácie klesla približne šesťkrát.

7.2.1 Stratégia alokácie a uvoľňovania pamäte

GC si pre každú generáciu drží vektor alokovaných miest. Alokované miesto je definované triedou `GCSpacePart`, v ktorej sa nachádza mutex pre synchronizáciu paralelnej alokácie, ukazovateľ na nasledujúcu alokovanú `GCSpacePart`, veľkosť alokovaného miesta, počet aktuálne použitých bytov a ukazovateľ na začiatok samotného miesta. Objekt rovnako implementuje rozhranie pre alokáciu miesta využívane GC pri alokácii. To vráti buď ukazovateľ na začiatok alokovaného miesta, pričom sa zväčší počet využitých bajtov, alebo `NULL`, ak v danej `GCSpacePart` nie je dostatok miesta pre alokovanie požadovanej veľkosti. Na začiatku behu simulácie sa alokuje jedno miesto s vopred stanovenou veľkosťou. V prípade, ak nie je dostatok mieta v `GCSpacePart`, ktorá sa aktuálne používa pre alokáciu, GC vytvorí novú s veľkosťou maxima z veľkosti objektu a dvojnásobku veľkosti predošlej `GCSpacePart`. Novo alokovanú časť následne previaže s predošlou cez ukazovateľ na ďalšiu časť a vloží do vektoru pre danú generáciu (algoritmus 7.1). Pri behu cyklu GC sa alokuje nová `GCSpacePart` pre *tospace*, ktorá má veľkosť súčtu aktuálne alokovaných `GCSpacePart`, alebo polovicu tohto miesta, ak je využitá len štvrtina aktuálne alokovaného miesta. Štvrtinu uvažujem preto, aby sa nemusela alokovať nová `GCSpacePart` hneď pri vytvorení nového objektu, ale aby toSpace malo istú rezervu miesta pre nové objekty. Táto stratégia je popísaná v algoritme 7.2.

```
/* V triede GarbageCollector */

void* GCAlloc(unsigned size):
    /* Ziskanie posledneho GCSpacePart z vektora spaces */
    GCSpacePart lastPart = spaces[spaces.size()-1]
    void* allocatedPtr = lastPart.alloc(size)
    if (allocatedPtr != NULL):
        return allocatedPtr
    /* nie je dostatok miesta */
    unsigned toAllocSize = max(2*getAllocated(), size)
    GCSpacePart newSpacePart = new GCSpacePart(toAllocSize)
    /* previazanie ukazatelov zoznamu */
    lastPart.setNextPart(newSpacePart)
    spaces.push(lastPart)
    return lastPart.alloc(size)

/* V triede GCSpacePart */

/* Konštruktor */
GCSpacePart(unsigned size):
    used = NULL
    allocatedSize = size
    nextPart = NULL
    allocatedSpaceBeginAddress = malloc(size)

void* alloc(unsigned size):
    /* zamedzenie paralelného alokovania kvôli posunu ukazovateľa */
    mutex.lock()
    if allocatedSize - used < size:
        mutex.unlock()
        return NULL
    used += size
    mutex.unlock()
    /* vrátenie adresy prvého bajtu alokovaného miesta zväčšenú o
       hodnotu used */
    return allocatedSpaceBeginAddress + used
```

Algoritmus 7.1: Alokácia nových objektov mutátorom. Mutátor volá metódu GCAlloc, ktorá mu vráti ukazovateľ na začiatok alokovaného miesta, kam má uložiť nový objekt.

```
unsigned getTospaceAllocSize():
    /* Vráti veľkosť alokovaného miesta, ktorá je súčtom alokovaného
       miesta v každom GCSpacePart vo vektore alokovaných
       GCSpacePart */
    unsigned toAlloc = getAllocated()
    /* Vráti veľkosť využitého miesta, ktorá je súčtom využitého
       miesta v každom GCSpacePart vo vektore alokovaných
       GCSpacePart */
    unsigned used = getUsed()
    /* zamedzenie paralelného alokovania kvôli posunu ukazovateľa */
    if used < toAlloc/4:
        toAlloc = toAlloc / 2
```

```
return toAlloc
```

Algoritmus 7.2: Stratégia získania veľkosti miesta toSpace na začiatku cyklu GC.

7.3 Zavedené rozšírenia správania sa objektov alokovaných na halde

Pre beh GC bolo potrebné ešte stanoviť spoločné správanie objektov alokovaných na halde. To som zabezpečil implementáciou triedy `GCObject`, z ktorej dedí každý objekt alokovaný na halde. Táto trieda obsahuje číslo instancie vytvoreného objektu, adresu, kam bol daný objekt pri cykle GC prekopírovaný, adresu mutexu, ktorý synchronizuje data-race v cykle GC pri prvotnom prístupe vlákna k objektu, aby objekt nebol viackrát skopírovaný (ilustruje digram 7.1) a počet prežitých cyklov GC. Rovnako implementuje virtuálnu metódu `getReferences`, ktorá vráti vektor adries uloženia referencií na iné `GCObject` z daného objektu (`GCObject**`). Vrátenie adresy uloženia ukazovateľa je dôležité pre možnosť prepisu na adresu nového uloženia pri cykle GC. Ďalej je tu implementovaná virtuálna metóda pre vrátenie veľkosti daného objektu (`getSize`), metódy pre prácu s mutexom a metóda pre nastavenie `forwardingAddress`. Veľmi podstatnou metódou je metóda `cloneTo`, ktorá skopíruje objekt na požadované miesto v pamäti. Táto metóda je priamo volaná GC pri kopírovaní objektu. Z hľadiska objektov, ktoré dedia od `GCObject` je teda podstatné implementovať kopírovací konštruktor, ktorý táto metóda využíva. Rovnako je potrebné preťažiť metódy `getSize` tak, aby vracala veľkosť aktuálneho objektu a `getReferences`, kde sa k `parent::getReferences()` pridajú aj adresy uloženia ukazovateľov na objekty z nového objektu.

7.3.1 Problém virtuálnych metód

Je dôležité spomenúť, prečo metóda `copyTo` využíva kopírovacie konštruktory objektov miesto toho, aby ich kopírovala napríklad pomocou funkcie memcpy. Objekty PNtalku obsahujú virtuálne metódy. Tie sú vo väčšine prekladačov implementované virtuálnou tabuľkou metód. V štandarde C++ ale nie je uvedené umiestnenie tejto tabuľky v objekte, ale ani to, že virtuálne metódy musia byť implementované práve pomocou tejto tabuľky. Pri prekopírovaní objektu pomocou `memcpy` sa prekopíruje do novej oblasti objekt aj s touto tabuľkou, v ktorej ale ostávajú adresy metód starého objektu. Po uvoľnení starého fromspace teda tieto metódy vyvolávajú chybu typu segmentation fault. Bolo by síce možné tieto hodnoty v tabuľke prepísať, no tým by sa riskovala strata multiplatformnosti a schopnosti preložiť kód do funkčnej podoby rôznymi prekladačmi.

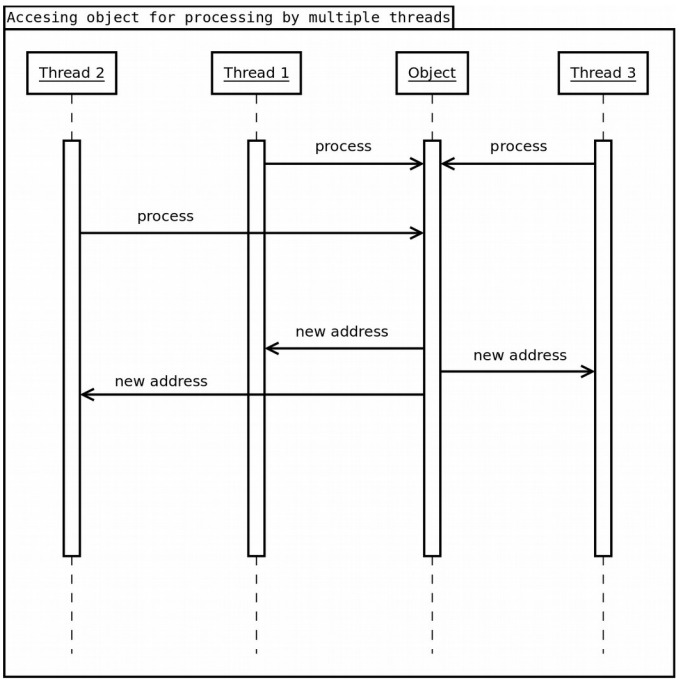


Diagram 7.1: Diagram prístupu vláken ku objektu. Ak objekt ešte nebol skopírovaný do toSpace, metóda `process` ho skopíruje a vráti adresu jeho nového umiestnenia. Ak bol objekt skopírovaný už skôr, metóda vráti jeho novú adresu. Prípad vyššie ukazuje, že túto metódu je potrebné synchronizovať. Vlákno 1 a 3 pristúpia naraz ku objektu, ktorý ešte nebol skopírovaný. Vlákno 1 uzamkne mutex skôr a začne objekt kopírovať. Medzitým zavolá na tento objekt metódu `process` ešte vlákno dva. Po skopírovaní vráti objekt jeho novú adresu všetkým vláknam. Bez synchronizácie by všetky tri vlákna objekt skopírovali na nové miesto.

7.3.2 Problém kopírovania mutex objektov

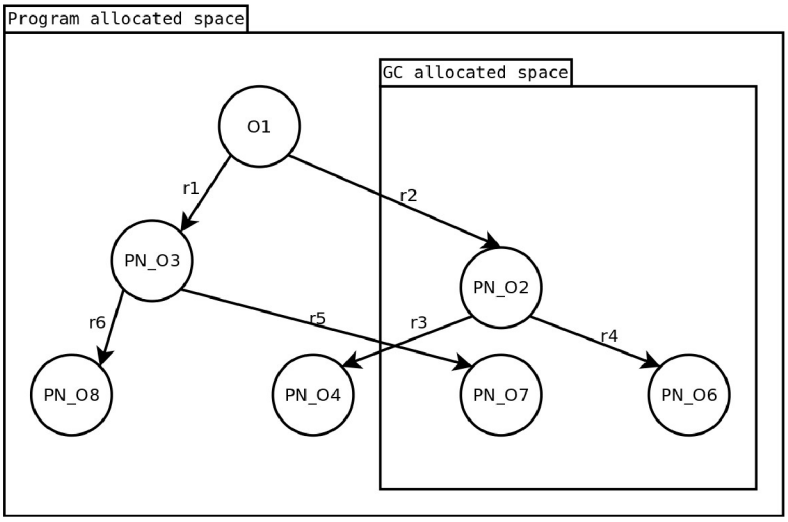
Ako som uviedol vyššie, je potrebné mať pre každý objekt istú synchronizáciu. Každý objekt PNtalk musí teda obsahovať mutex pre synchronizáciu prístupu k nemu. Mutex je objekt, ktorý sa nedá kopírovať, resp. premiestňovať. Bolo teda potrebné ukladať mutexy niekde inde a do objektov uložiť iba adresu mutexu. Mutexy som preto uložil do kolekcie list, ktorá umožňuje jednoduché uvoľňovanie mutexov patriacich jednotlivým objektom, ktoré boli uvoľnené v cykle GC bezo zmeny ich adresy. Aby bola možná paralelná *sweep* fáza pre mutexy, používam *n* zoznamov mutexov, pričom sa jednotlivé zoznamy z poľa cyklicky striedajú pre uloženie nového mutexu. Pre uloženie mutexov by kolekcie typu vektor ani neboli použiteľné, keďže pri realokácii nie je možné mutex prekopírovať na iné miesto. Tieto mutexy preto spravujem pomocou upraveného *mark-sweep* algoritmu v každom cykle GC. Každá položka zoznamu má teda okrem objektu mutex aj značku. Pri každom cykle GC sa pri prechode jednotlivými objektami nastavujú značky pre im odpovedajúce mutexy a následne po dobehnutí kopírovacieho algoritmu nastáva sweep fáza pre mutexy spočívajúca v paralelnom prejdení zoznamu mutexov a uvoľnení tých, ktoré nie sú označené.

7.4 Popis behu GC

Začiatok cyklu GC pre mladšiu generáciu je iniciovaný podľa počtu iterácií simulačnej slučky podľa nastavených parametrov na konci každej *k*n*-tej iterácie. Medzi každý *k*m*-tý a *k*(m+1)* beh GC je vložený cyklus pre staršiu generáciu. Ako počiatočná množina koreňových objektov boli zvolené referencie na prechody uložené v objekte `PNsimulation`. Následne je implementovaný algoritmus z prílohy A.9. Kopírovací algoritmus, podobne ako pri Smalltalkovskej implementácii, začína jedným vláknom. Následne, ak je v pracovnom zozname viac položiek, ako vláken, tento zoznam sa rozdelí rovnomerne medzi všetky vlákna a začína sa paralelný beh GC. Dôležité je podoknúť, že metóda `copy` kopíruje objekt pri zbere mladšej generácie buď do toSpace, alebo pri dosiahnutí potrebného počtu cyklov do staršej generácie. V cykle pre mladšiu generáciu je potrebné zaistiť, že metóda `forward` vráti pre každý objekt zo staršej generácie jeho aktuálnu polohu a rovnako pri cykle staršej generácie to isté pre objekty z mladšej generácie. *Marking* fáza pre jednotlivé mutexy je riešená pri prístupe do každého objektu, pričom je dobré spomenúť, že nie je podstatné koľkokrát je daný mutex označený v jednom cykle – viacnásobným označením mutexu nevzniká žiaden problém. Následne sú zoznamy mutexov paralelné prejdené vláknami, ktoré odstránia zo zoznamu neoznačené mutexy.

7.5 Externé referencie

Problém externých referencií v prípade C++ obsahuje o niečo viac podproblémov ako v prípade Smalltalk. Keďže programátor pracuje priamo s umiestnením objektov v pamäti, je potrebné riešiť štyri prípady. Ak interná referencia ukazuje na objekt, ktorý je uložený v rámci alokovaného miesta GC, ak externá referencia ukazuje na takýto objekt, ak interná referencia ukazuje na objekt uložený mimo alokovaného miesta GC a ak externá referencia ukazuje na takýto objekt. To ilustruje obrázok 7.3. Keďže C++ využíva explicitnú prácu s pamäťou, môj návrh sa tiež držal tejto logiky. Tento prístup je v súlade s explicitnou správou pamäti, ktorá je v C++ používaná. Pri vytvorení externého ukazovateľa sa zavolá `registerExtern`, ktorá vloží ukazovateľ na tento externý ukazovateľ do zoznamu `root`, ktorý sa predáva GC, ako zoznam počiatočných objektov. Pri cykle GC sú tak všetky takéto ukazovatele pokladané za vstupné body programu. Prípad, ak objekt je uložený mimo alokovaného miesta GC, je ošetrený v metóde `process`. Riešenie spočíva v teste, či adresa uloženia objektu spadá do alokovaného *fromspace* alebo *tospace*. Ak sa jedná o objekt uložený mimo takéhoto miesta, daný objekt sa uloží do worklistu, no nehýbe sa ním. Je dobré samozrejme zvalidovať, či sa jedná o stále platný ukazovateľ, ktorý neostal v rámci simulácie po objekte, ktorý bol zozbieraný a nebol nastavený na hodnotu `NULL`. Vo vytvorenej implementácii sa preto každý ukazovateľ na potenciálne žijúci objekt, ktorý je uložený mimo aktuálneho *fromspace* a *tospace* overuje, či sa nachádza v zozname externých referencií, ktorý vzniká pri volaní metódy `registerExtern`. Ak sa tam testovaný ukazovateľ nenachádza, nejedná sa o validnú referenciu a je ignorovaná. Zrušenie takéhoto objektu je následne podmienené zavolaním metódy `unregisterExtern`, ktorá daný objekt uvoľní zo zoznamu externých objektov a zoznamu root objektov. Úprava metódy `process` je popísaná v algoritme 7.3.



Obrázok 7.3: Rôzne typy referencií. O1 nie je objekt PNtalk, teda referencie z neho na objekty prostredia PNtalk sú externé. Ostatné objekty (začínajúce prefixom PN) sú objekty prostredia PNtalk. GC musí správne pracovať aj s vyššie uvedeným grafom objektov. Referencie r1 a r2 sú externé referencie na objekt uložený mimo oblasti spravovanej GC (r1) a na objekt spravovaný v rámci oblasti GC (r2). Referencie r3 a r4 sú interné referencie na objekt uložený mimo oblasti spravovanej GC (r3) a na objekt uložený v oblasti spravovanej pomocou GC. Je potrebné si uvedomiť, že v rámci cyklu musí GC prechádzať aj objekty alokované mimo priestor ním spracovaný (také objekty nesmie premiestniť), pretože môžu mať referencie na objekty uložené v rámci priestoru, ktorý spravuje (prípad r5).

```
void process(GCObject** object, queue worklist, vector<GCSpacePart>
  generation)
    GCObject* fromRef = *object
    /* Test na validitu adries, nevalidné adresy sú mimo
       alokovaných generácií, resp. tospace (napr. NULL) */
    if ((inFromspace(object)) or inTospace(object)):
        /* bráni viacnásobnému skopírovaniu */
        fromRef->lock()
        /* zároveň marking fáza pre mutexy, viacnásobné označenie
           tu nie je problém, lebo metóda forward zabráni
           viacnásobnému prechodu grafu objektov */
        fromRef->markMutex()
        /* nastavenie referencie na novú adresu objektu */
        *object = forward(fromRef, worklist, generation)
        fromRef->unlock()

    /* Externý objekt */
    else isValidReference(object):
        fromRef->lock()
        if not worklist.contains(object):
            worklist.push_back(object)
```

```
fromRef->unlock()
```

Algoritmus 7.3: Riešenie externých referencií v C++ implementácii.

7.6 Zhodnotenie riešenia a možné optimalizácie

Implementácia aj v tomto prípade využíva možnosti paralelného hardvéru. Tri miesta v algoritme si vyžadujú sekvenčný prístup. Prvým miestom je alokácia nového miesta v `GCSpacePart`, no tá spočíva len v inkrementácii ukazovateľa. Toto riešenie ponúka niekoľko optimalizácií behu mutátora, ktorými sa snaží kompenzovať zdržania *garbage collectingom*. Druhé takéto miesto je pri prístupe k privátnym frontám kopírovacích vláken GC. Takýto paralelný prístup opäť nie je častý. Tretie miesto je pri prístupe k objektu pri cykle GC. Pozastavenie vlákna tu môže byť relatívne dlhé, v prípade, že sa jedná o veľký objekt, ktorý je kopírovaný iným vláknom, no pri množstve objektov, ktoré PNtalk generuje a pri ich charaktere, že objekty sú väčšinou malých veľkostí sa ale nepredpokladá častý výskyt takejto udalosti.

Je potrebné uviesť problém implementácie simulačnej knižnice, ktorý znemožňuje efektívne uvoľňovanie sietí objektov. Pri vytvorení siete, či už metódy alebo objektu, sa pridajú všetky prechody tejto siete do vektoru možných prechodov. V prípade siete metód sa uvoľnenie týchto registrovaných prechodov deje pri zániku siete, čiže dosiahnutí miesta return, čo explicitne garantuje, že daná sieť má zaniknúť a nikdy v budúcnosti už nebude žiaden jej prechod uskutočnený. V prípade siete objektov ale nie je možné zistiť zánik siete, keďže žiadne miesto typu return taká sieť neobsahuje. Simulačná knižnica ani neukladá referenciu na takto vytvorený objekt po registrovaní jeho prechodov. Implementácia simulačnej knižnice teda neumožňuje zistiť, kedy takáto sieť má byť zneplatnená a ktoré prechody teda majú byť uvoľnené. Riešenie tohto problému si vyžaduje hlbšiu zmenu logiky simulácie, čo je mimo rozsah tejto práce.

Priestor pre ďalší vývoj je podobne ako v kapitole 6.2.5 a implementácií v Smalltalk v efektívnejšom využívaní bitov premenných potrebných pre beh GC a v navrhnutí dátovej štruktúry pre uloženie spracovávaných položiek pre jedno kopírovacie vlákno. V tomto prípade je ale možné tieto problémy riešiť efektívnejšie ako v prípade implementácie v Smalltalku.

8 Testovanie a stanovenie optimálnych parametrov

V tejto kapitole sú popísané testovacie príklady, ktoré som v rámci práce vytvoril, ďalej je tu popísané meranie a stanovenie optimálnych parametrov behu GC a nakoniec sú uvedené porovnania výkonnosti a pamäťovej náročnosti oproti implementácie bez GC.

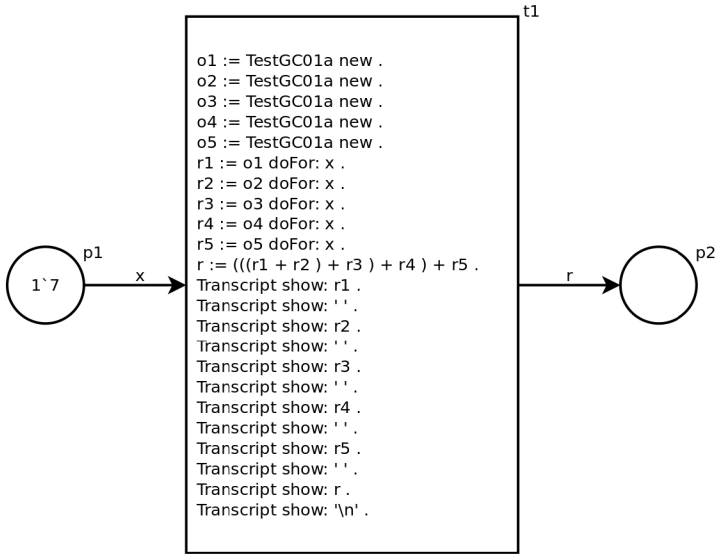
8.1 Popis príkladov

V rámci práce som vytvoril niekoľko príkladov pre testovanie, odladenie a ukážku práce vytvorených GC. Týmito príkladmi som chcel pokryť rôzne scenáre vytvárania nových objektov a na základe toho nájsť vhodné parametre GC. Keďže externé referencie nie sú modelovateľné nezávisle na implementácií, bol som nútený vytvoriť rozličné príklady pre C++ a Smalltalk implementáciu. Tieto príklady, spolu s príkladom na synchrónne porty, boli vytvorené čiste na overenie správnosti, a neboli použité pre stanovenie optimálnych parametrov behu, keďže vychádzajú z implementácií iných príkladov, ktoré boli pre výkonnostné testy použité.

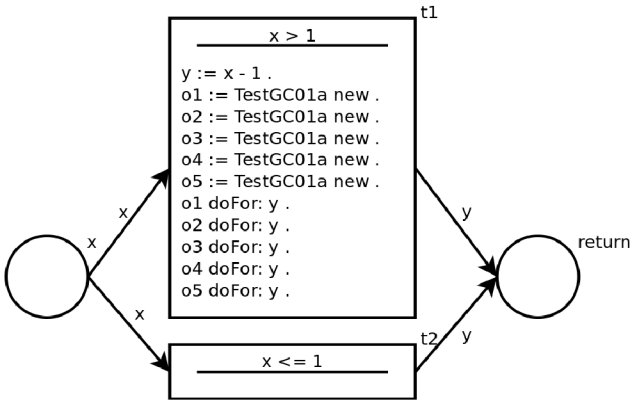
8.1.1 Garbage collecting sietí metód

Prvý príklad sa zameriava na siete vytvorené pri volaní metód objektov. V tomto príklade je tvorený strom objektov s exponenciálne rastúcim počtom týchto objektov. Každý objekt v rámci zavolanej metódy vytvorí 5 ďalších objektov, v ktorých volá metódu, ktorá vytvorí 5 ďalších objektov. Počet takýchto zanorení je nastavený v prvom objekte. Sieť metódy je zobrazená na obrázku 8.1 . Zdrojový kód príkladu sa nachádza v prílohe B1.

a)



b)

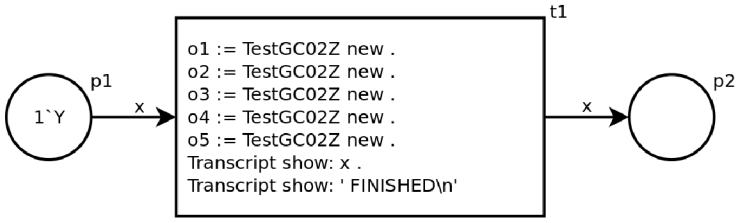


Obrázok 8.1: Sieť príkladu TestGC01. Sieť a) tvorí objektovú sieť triedy TestGC01 a b) tvorí sieť metódy doFor: s triedy TestGC01a.

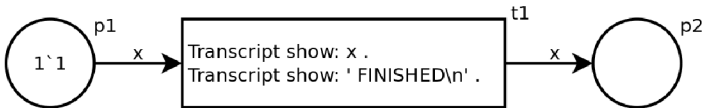
8.1.2 Garbage collecting sietí objektov

Druhý príklad je modifikáciou prvého. Tentokrát som sa zameral ale na vytváranie sietí objektov. Každý objekt vytvorí v rámci svojej siete 5 ďalších objektov až po stanovený počet zanorení. Sieť takýchto objektov je znázornená na obrázku 8.2. Zdrojový kód príkladu sa nachádza v prílohe B.2.

a)



b)



Obrázok 8.2: Sieť objektov z príkladu TestGC02. Sieť a) zastupuje triedy {TestGC02, TestGC02a, TestGC02b, TestGC02c, TestGC02d}. V názve TestGC02Z znamená 'Z' nasledujúcu volanú triedu, teda Z(TestGC02) = TestGC02a, Z(TestGC02a) = TestGC02b, Z(TestGC02b) = TestGC02c Z(TestGC02c) = TestGC02d a Z(TestGC02d) = TestGC02e. Podobne Y v mieste p1 Y(TestGC02) = 6, Y(TestGC02a) = 5, Y(TestGC02b) = 4, Y(TestGC02c) = 3 a Y(TestGC02d) = 2. Sieť b) predstavuje sieť objektu TestGC02e.

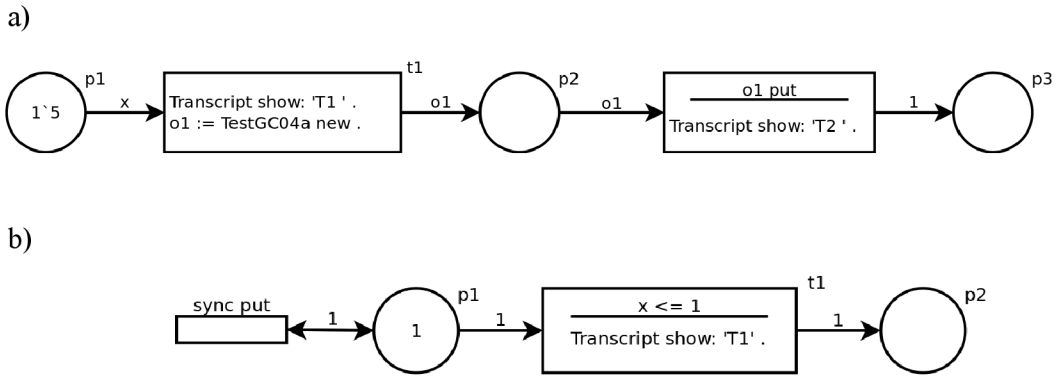
8.1.3 Garbage collecting sietí objektov a metód

Tretí príklad sa mierne líši v C++ a Smalltalk implementácii. V C++ implementácii som pre ladenie využil problém, uvedený v kapitole 7.6, a to, že C++ implementácia nedokáže uvoľnovať siete objektov. Tento fakt ale pozitívne vplýva na odladenie GC pre veľké simulácie s mnohými objektami. V tomto príklade sú striedavo vytvárané objektové siete a siete metód a to podobnou logikou ako v

predošlých príkladoch. Tento príklad simuluje bežný beh programov. Pre rozsiahlosť a rozdiel v zdrojových kódoch pre obe implementácie je uvedený len na CD v prílohe C.

8.1.4 Garbage collecting sietí so synchrónnymi portami

Štvrtý príklad sa sústreďuje na využívanie synchrónnych portov. Slúži len na overenie funkčnosti simulátoru. Sieť prvotného objektu v jednom prechode vytvorí sieť druhého objektu a následne v ďalšom prechode pomocou synchrónneho portu overuje stav vytvoreného objektu. Siete objektov a metód sú zobrazené na obrázku 8.3. Zdrojový kód príkladu sa nachádza v prílohe B.4.



Obrázok 8.3: Sieť príkladu TestGC04. Sieť a) tvorí objektovú sieť triedy TestGC04 a b) tvorí objektovú sieť triedy TestGC04a so synchrónnym portom

8.1.5 Garbage collecting externých referencií - C++

Pre overenie funkčnosti externých referencií som vytvoril dva príklady. Oba sú upravené verzie príkladu 8.1.2. Počiatočný objekt simulácie vytvorím ako externý, v prvom prípade alokovaný vrámci priestoru spravovaného GC a v druhom mimo. Po zbehnutí simulácie sa následne k objektu pristupuje a vypíšu sa o ňom informácie. Objekt nesmel byť po skončení simulácie uvoľnený. Tieto príklady pre rozsiahlosť zdrojových kódov uvádzam v príklade na CD v prílohe C.

8.1.6 Garbage collecting externých referencií – Smalltalk

Tento príklad je upravený príklad z kapitoly 8.1.2 tak, že k simulácií bol vytvorený externe nový počiatočný objekt a registrovaný v simulácií. Príklad ukazuje, že simulácia po registrovaní tohto objektu beží aj pre objekty ním vytvorené, no po odstránení externej referencie na tento objekt GC uvoľní všetky objekty ním vytvorené. Tento príklad pre duplicitnosť zdrojových kódov s príkladom 8.1.2 uvádzam v príklade na CD v prílohe C.

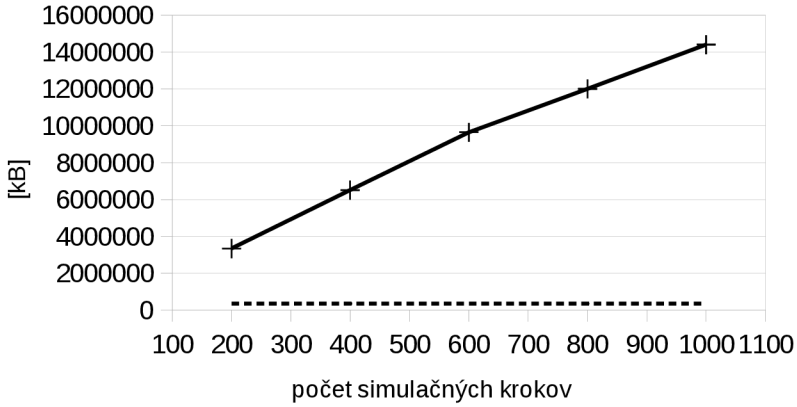
8.2 Stanovenie optimálnych parametrov

V prvom rade bolo potrebné zvoliť referenčnú architektúru. Keďže som chcel, aby boli merania čo najpresnejšie zvolil som beh GC na dvoch vláknach. Referenčný procesor, na ktorom som výkon meral (typ procesoru) má 2 fyzické jadrá s podporou hyperthreadingu. To umožňuje, aby obe vlákna GC bežali paralelne. Pri zvolení viac vláken by mohli vznikať časté štrukturálne konfliky pre nedostatok zdrojov ponúknutých referenčným hardvérom. Všetky merania som robil na OS Ubuntu 15.10 64-bit. Ako referenčný hardvér som použil Intel Core i7-4510U 2GHz a 16 GB RAM. Každé meranie som opakoval trikrát a uvádzam priemerné hodnoty meraní.

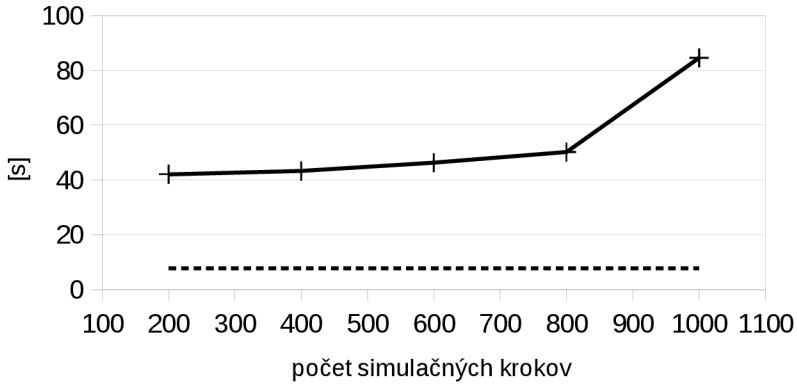
8.2.1 Implementácia v C++ a jej zhodnotenie

Meranie spočívalo vo vytvorení skriptu, ktorý spúšťal opätovne simulácie s rôznymi parametremi. Pre počet cyklov som zvolil granularitu zmeny na 200 cyklov a pre počet cyklov GC pre presunutie objektu do staršej generácie, rovnako ako pre rozdiel v cykloch GC medzi zberom mladšej a staršej generácie, granularitu s jednotkou 2.

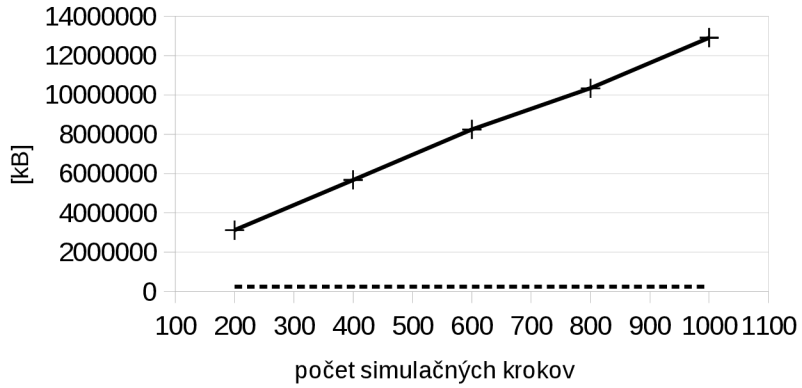
V grafoch 8.1 až 8.6 uvádzam časové a pamäťové nároky príkladov z kapitol 8.1.1 až 8.1.3 pri zmene počtu simulačných krokov medzi cyklami GC. Z grafov 8.1, 8.3 a 8.5 je vidieť, že nárast veľkosti objektov je závažným problémom tejto implementácie. Treba si uvedomiť, že napríklad pridaním jedného ukazovateľa na 64 bitovej architektúre narastie veľkosť objektu o 8b. Grafy 8.1 a 8.3 ďalej ukazujú, že problém popísaný v kapitole 7.6 s nemožnosťou efektívneho uvoľňovania sietí objektov je závažným nedostatkom implementácie simulátoru. Tento problém spôsobuje, že objektové siete ostávajú v pamäti do konca simulácie. Ako je vidieť na grafe 8.5, pri využívaní čiste objektov metód takýto problém nevzniká a implementácia s GC má výrazne nižšiu pamäťovú náročnosť ako implementácia bez GC. Pre odstránenie tohoto problému by bola potrebná hĺbková zmena implementácie virtuálneho stroja, čo presahuje rozsah tejto práce.



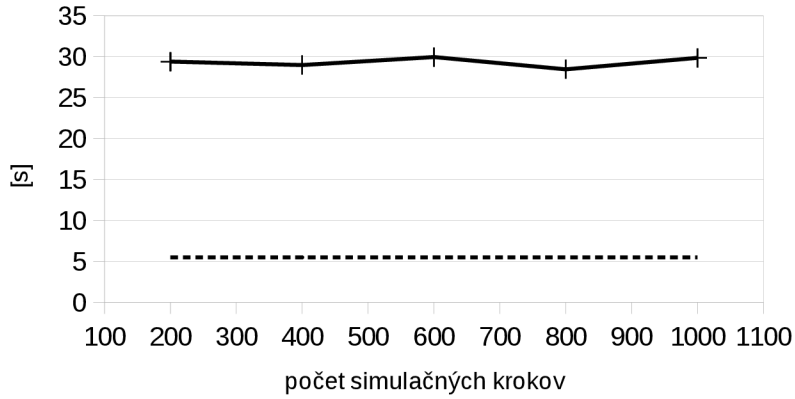
Graf 8.1: Pamäťová náročnosť simulácie (príklad z kapitoly 8 .1.3) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.



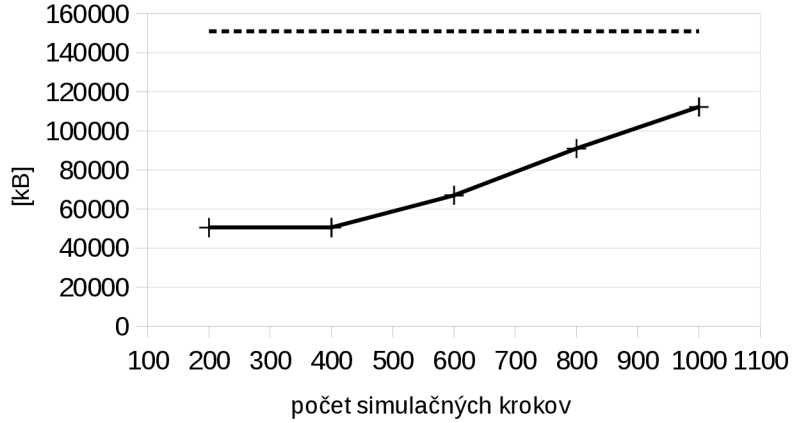
Graf 8.2: Časová náročnosť simulácie (príklad z kapitoly 8 .1.3) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.



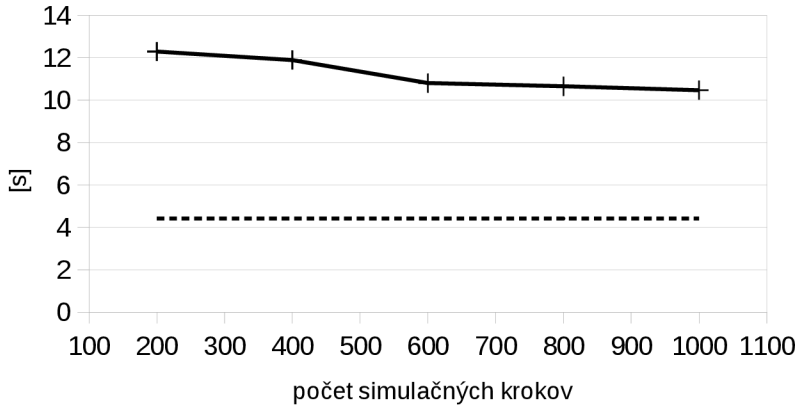
Graf 8.3: Pamäťová náročnosť simulácie (príklad z kapitoly 8 .1.2) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.



Graf 8.4: Časová náročnosť simulácie (príklad z kapitoly 8.1.2) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.



Graf 8.5: Pamäťová náročnosť simulácie (príklad z kapitoly 8 .1.1) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.



Graf 8.6: Časová náročnosť simulácie (príklad z kapitoly 8 .1.1) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC. Čiarkovaná čiara znázorňuje referenčnú náročnosť bez GC.

Na grafoch 8.2, 8.4 a 8.6 môžeme vidieť aj ďalší problém, a to je dva až šesťnásobné zníženie výkonnosti oproti referenčnej výkonnosti. Tento problém je podmienený zložitejším procesom vytvárania nových objektov a samozrejme behom samotného GC. Pri veľkom počte objektov v pamäti ich kopírovanie pri cykle GC zaberá značnú časť procesorového času behu simulácie. To je vidieť hlavne na grafe 8.2 a 8.4, kde boli príklady, ktoré si držali veľký počet objektov v pamäti počas celej simulácie. Tento problém úzko súvisí s problémom implementácie popísaným v 7.6.

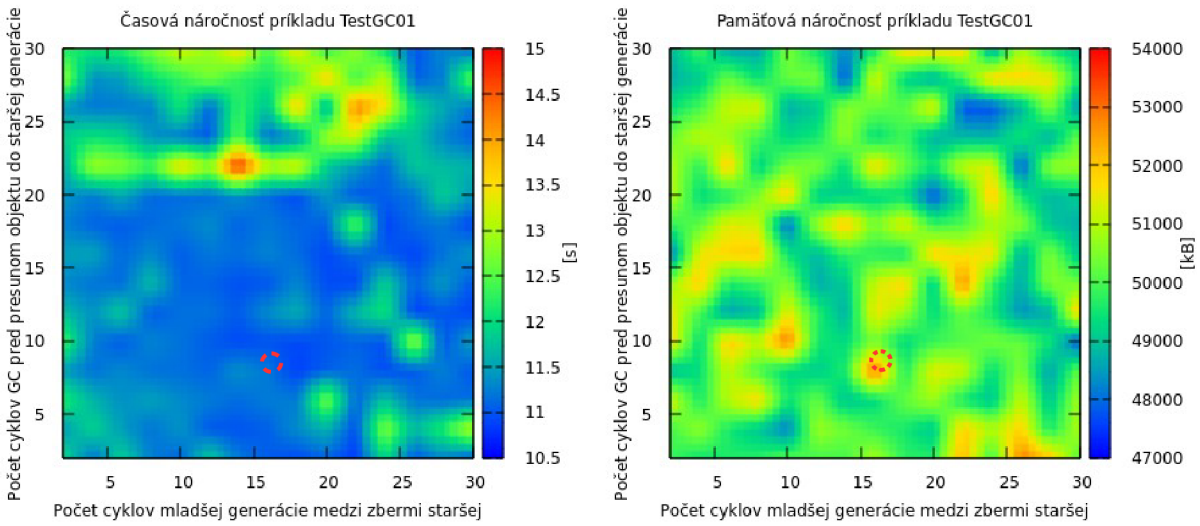
Zaujímavosťou ale je, že pri častejšom volaní GC sa znižuje časová náročnosť, jako je vidieť na grafe 8.2. To je spôsobené hlavne cache pamäťami a dobrou lokalitou dát pri častejšom behu GC. Veľký nárast časovej lokality medzi 800 a 1000 simulačnmi krokmi je spôsobený tým, že pri pamäťovej náročnosti pre 1000 simulačných krokov medzi cyklami GC sa začalo swapovať na disk.

Ďalším zaujímavým pozorovaním z porovnania grafov 8.3 s 8.5 a 8.4 s 8.6. je, že implementácia podobného príkladu pomocou volania metód objektov je pamäťovo aj výkonnostne efektívnejšia, ako implementácia využívajúca siete objektov. Príklad z kapitoly 8.1.1 predpokladá 7 zanorení a príklad z kapitoly 8.1.2 len päť, no napriek tomu je čas vykonávania 8.1.1 len o niečo málo vyšší. Tento fakt je opäť podmienený implementačným problémom so sieťami objektov, ktorý je spomenutý v 7.6.

Z grafou 6.1 je zrejmé, že pre použití viac ako 200 cyklov simulácie je pamäťová náročnosť neprijateľná. Zvyšné dva parametre boli stanovené podľa meraní, ktoré sú zobrazené na grafoch 8.7 až 8.9. Tieto merania ukazujú, že je výhodné nastaviť počet cyklov GC na presun do staršej generácie na 8, a rozdiel v počte cyklov GC medzi zberom generácií na 16.

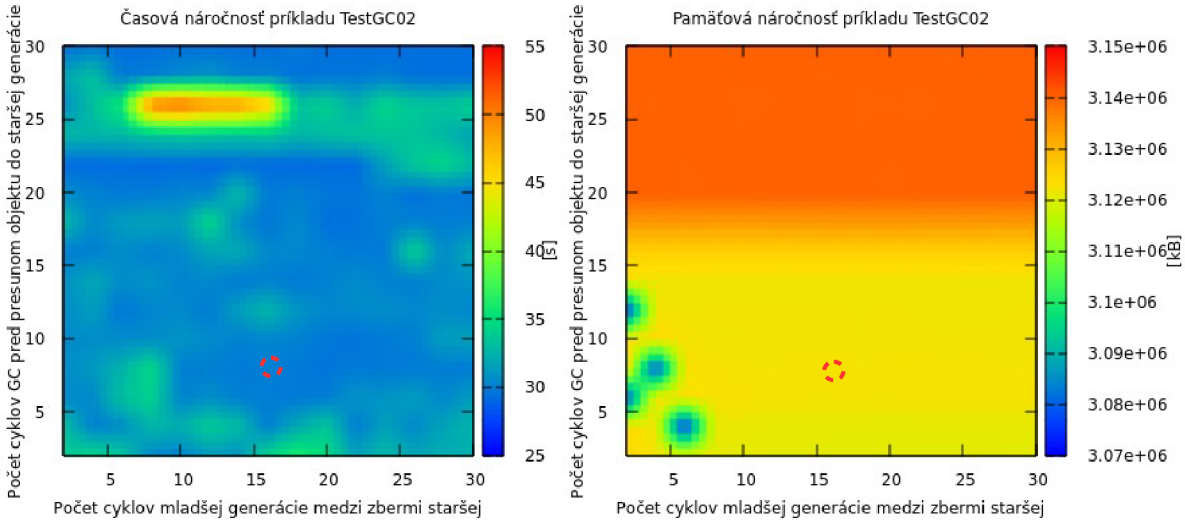
Zajímavým pozorovaním z grafov 8.8b a 8.9b je nárast pamäťovej náročnosti pri zvyšovaní počtu cyklov GC pre presun objektu do staršej generácie. Dôvodom je logika alokácie kolekcií. Pri naplnení kolekcie s veľký počtom prvkov sa pri realokácií zväčší alokovaný priestor kolekcie o väčšiu časť pamäte ako pri menších kolekciách. To potenciálne vedie k veľkej časti alokovanej pamäte, ktorá nie je využitá. Je dobré spomenúť, že množstvo objektov, ktoré reprezentujú značky, žije len niekoľko málo simulačných krokov a sú zozbierané hneď pri prvom cykle mladšej generácie (cyklus GC každých 200 simulačných krokov). Teda ani pri skorom presune objektov do staršej generácie táto generácia nenarastá príliš rýchlo.

a) b)



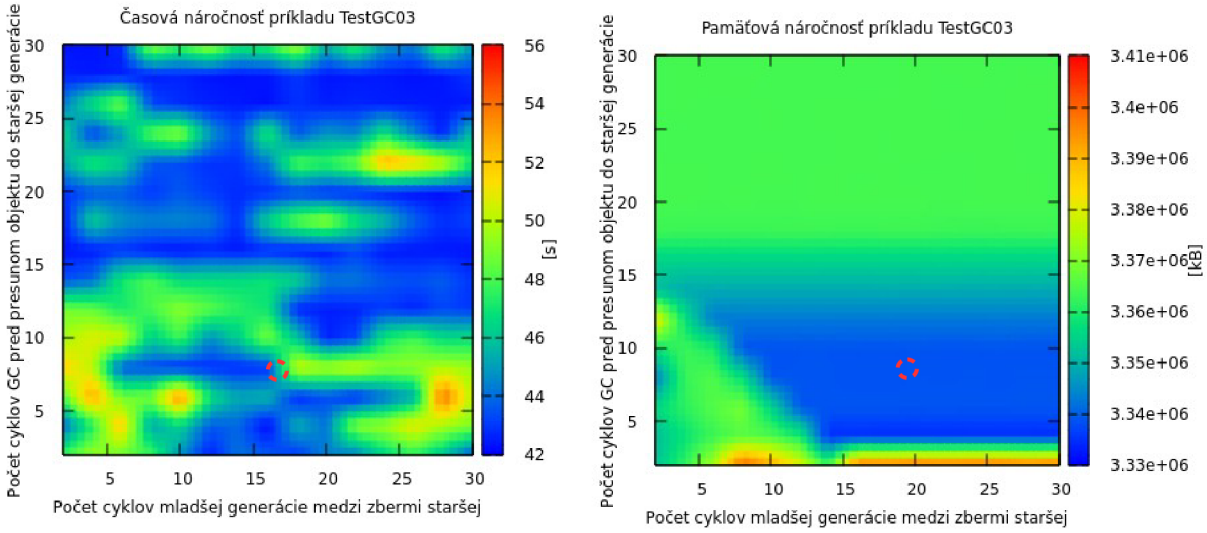
Graf 8.7: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.1 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

a) b)



Graf 8.8: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.2 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

a) b)



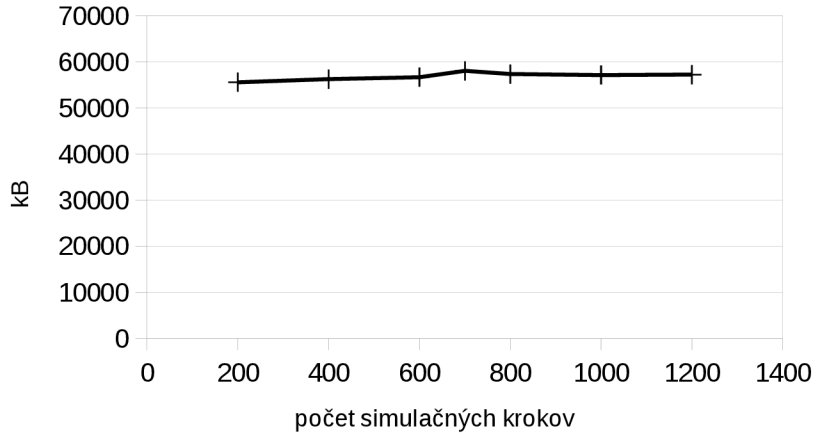
Graf 8.9: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.3 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

8.2.2 Implementácia v Smalltalk a jej zhodnotenie

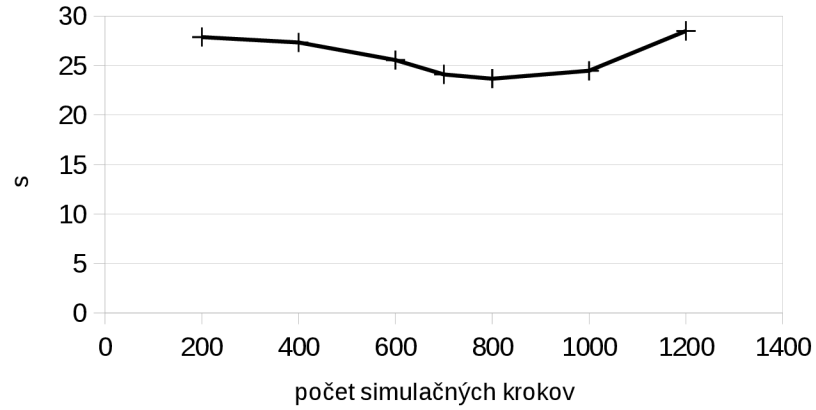
Pre ladenie tejto implementácie bolo potrebné znížiť nároky na počet objektov pri príkladoch. V príklade z kapitoly 6.1.1 pri počte 7 zanorení trvala simulácia s GC (200 simulačných krokov medzi cyklami GC) približne 11 minút a 30 sekúnd a pri príklade z kapitoly 6.1.2 47 minút a 2 sekundy. Aj v tejto implementácií sú príklady pracujúce so sieťami metód rýchlejšie ako príklady pracujúce so sieťami objektov. Je to spôsobené častejším testovaním uskutočniteľnosti prechodov pre siete objektov. Pamäťové nároky boli v prvom príklade len približne o 12 MB vyššie (52103 kB), no v druhom príklade 33-krát nižšie (92571 kB). Opäť sa tu odzrkadlil problém s neuvoľňovaním sietí objektov v C++ implementácii popísaný v kapitole 7.6. V prvom príklade som teda kvôli časovej náročnosti simulácie znížil počet zanorení na päť a v druhom na tri.

Pri testovaní tejto implementácie sa ukázal ešte jeden problém so synchronizáciou v simulácii, no ten pre krátkosť času už nebol riešený. Ako najslabšie miesto implementácie sa v testoch prejavilo falošné detekovanie ukončenia simulácie, kedy sa nadbytočne spúšťali cykly pre obe generácie a to spomaľovalo beh simulácie. Porovnania s implementáciou bez GC neboli možné pretože pôvodná implementácia zvládala maximálne prácu s cca 1500 objektami.

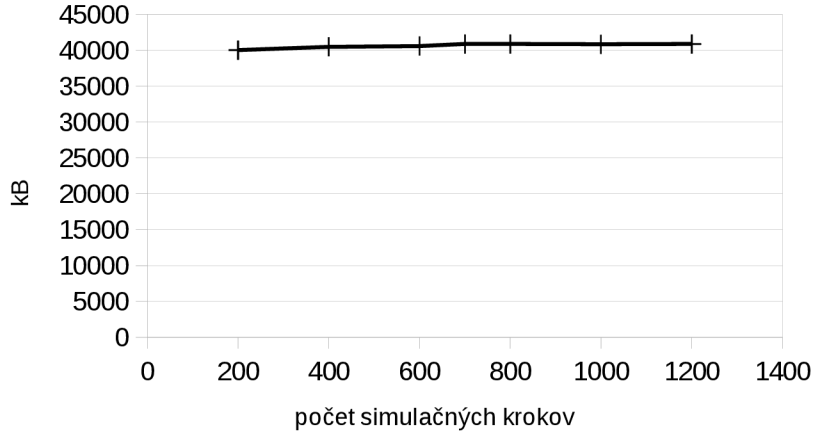
Ako v predošlom prípade som vykonal sériu testov s rôznymi parametremi GC. Opäť bolo potrebné stanoviť počet simulačných krokov medzi cyklami GC, počet cyklov GC, ktoré musí objekt prežiť na to, aby bol presunutý do staršej generácie a počet cyklov GC mladšej generácie medzi cyklami GC staršej generácie. Grafy 8.10, 8.12 a 8.14 ukazujú, že počet simulačných krokov medzi cyklami GC nemá výrazný vplyv na pamäťovú náročnosť, no grafy 8.11 8.13 a 8.14 ukazujú, že tento vplyv je výraznejší pri meraní časovej náročnosti. Pri menej simulačných krokoch je beh GC príliš častý, čo spomaľuje simuláciu. Pri viacerých krokoch zas ostáva v zozname `components` (kapitola 6.2.1) priveľa objektov, čo opäť spomaľuje simuláciu. Z vyššie uvedených grafov vyplýva, že optimálny počet simulačných krokov medzi zbermi generácií je 700.



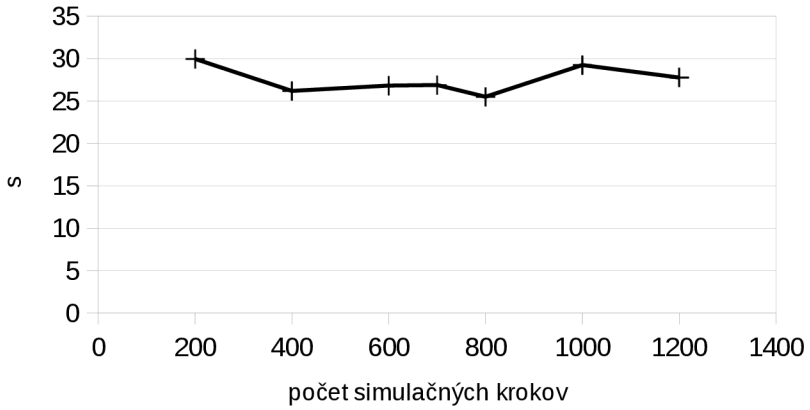
Graf 8.10: Pamäťová náročnosť simulácie (príklad 8.1.1) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.



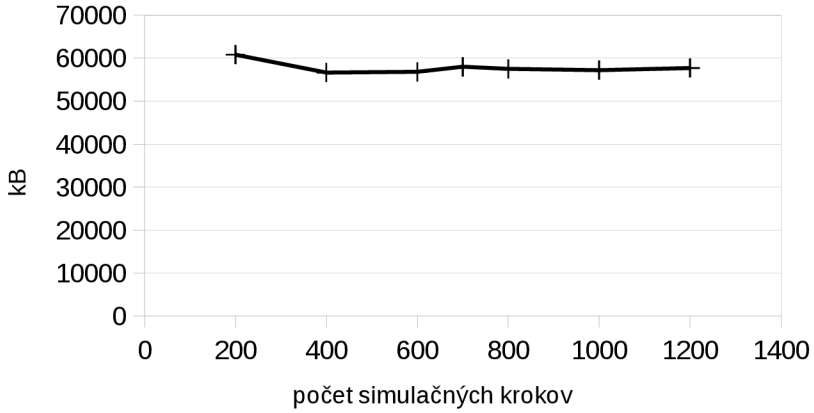
Graf 8.11: Časová náročnosť simulácie (príklad 8.1.1) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.



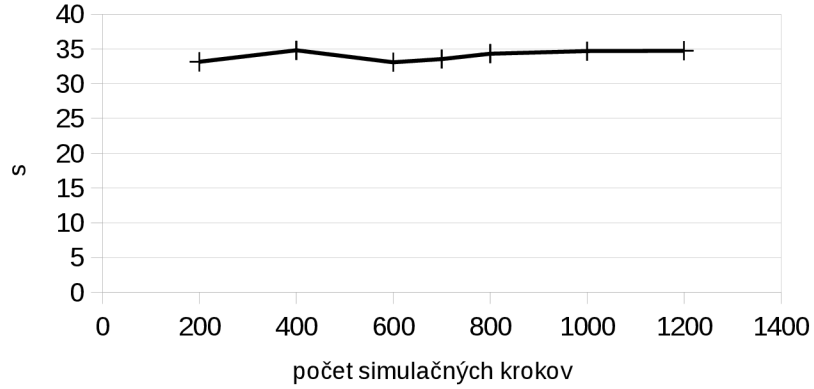
Graf 8.12: Pamäťová náročnosť simulácie (príklad 8.1.1) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.



Graf 8.13: Časová náročnosť simulácie (príklad 8.1.2) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.



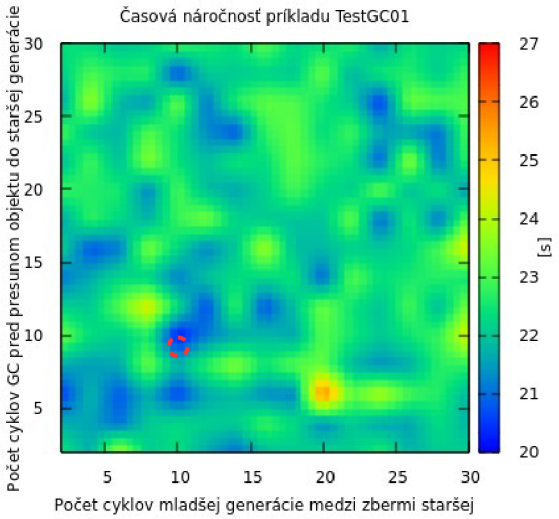
Graf 8.14: Pamäťová náročnosť simulácie (príklad 8.1.3) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.



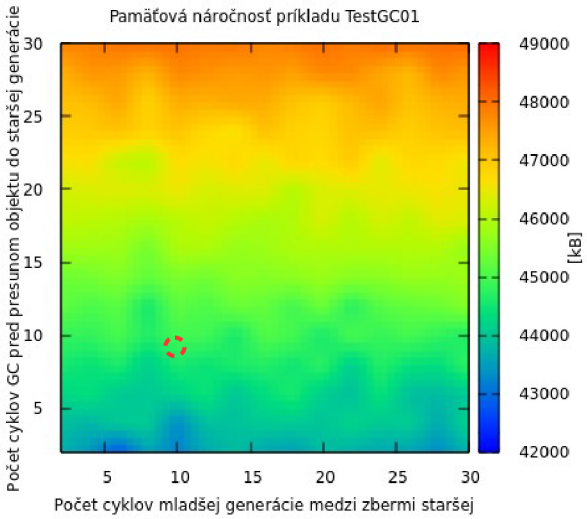
Graf 8.15: Časová náročnosť simulácie (príklad 8.1.3) v závislosti od počtu simulačných krokov medzi cyklami GC pri optimálnych ostatných parametroch GC.

Z grafov 8.16 b), 8.17 b) a 8.18 b) môžeme pozorovať, podobne ako 8.8 b) a 8.9 b), nárast pamäťovej náročnosti pri zvyšovaní počtu cyklov GC pre presun objektu do staršej generácie. Dôvod je podobný, ako pri C++ implementácii a to logika alokácie kolekcií (kapitola 8.2.1). Merania ale podľa grafov 8.16, 8.17 a 8.17 ukazujú, že je výhodné nastaviť počet cyklov GC na presun do staršej generácie na 8, a rozdiel v počte cyklov GC medzi zberom generácií na 10.

a)

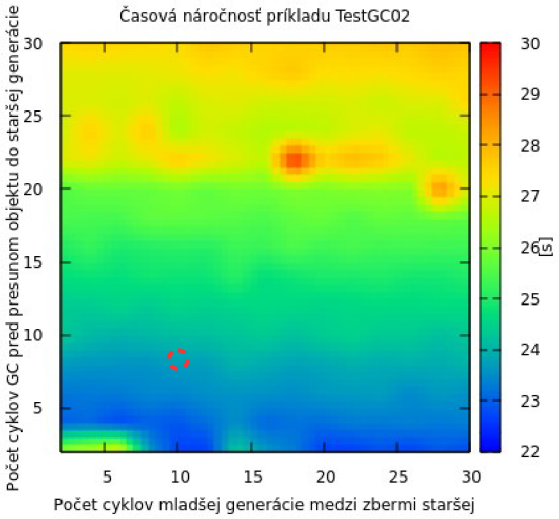


b)

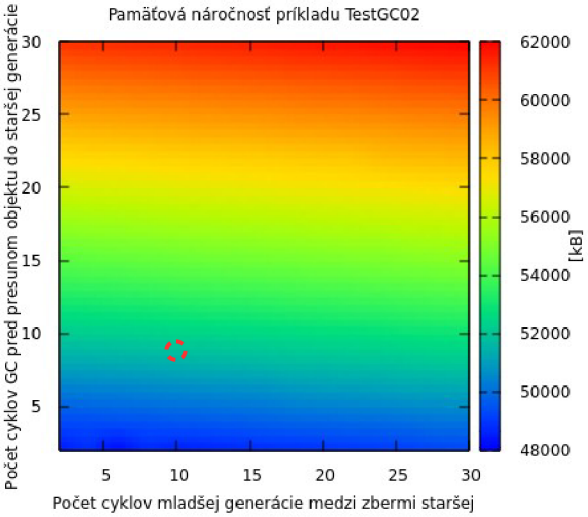


Graf 8.16: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.1 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

a)

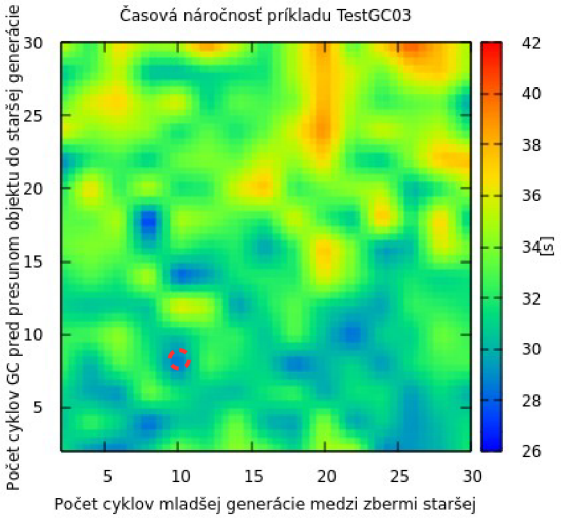


b)

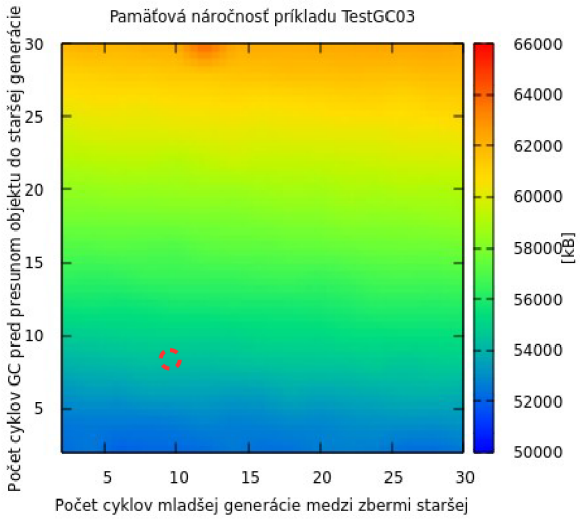


Graf 8.17: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.2 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

a)



b)



Graf 8.18: Graf časovej (a) a pamäťovej (b) náročnosti príkladu z kapitoly 8.1.3 v závislosti od počtu cyklov GC pred presunom objektu do staršej generácie a počtu cyklov mladšej generácie medzi zbermi staršej. Červený krúžok znázorňuje optimálne parametre.

9 Záver

V tejto práci som predstavil jednotlivé prístupy v problematike automatickej správy pamäte. Popísal som najznámejšie rodiny algoritmov, a to *mark-sweep*, *mark-compact*, kopírovacie algoritmy a počítanie referencií. Pre každú rodinu som zhodnotil ich pozitívne a negatívne vlastnosti. Zo začiatku som uviedol sekvenčné *stop-the-world* verzie. Následne som popísal ich paralelné a konkurenčné verzie. Zameral som sa aj na celkovú problematiku paralelných a konkurenčných algoritmov a na možné riešenia jednotlivých problémov. Ďalej som predstavil jazyk PNtalk, ktorý ponúka možnosť abstrakcie pri tvorbe modelov a zachováva jednoduchú formálnu analyzovateľnosť Petriho sietí a možnosť simulácie. Nakoniec som s použitím vyššie popísaných algoritmov a prístupov navrhol a implementoval garbage collector pre obe implementácie virtuálneho stroja jazyka PNtalk. V tejto práci som tiež navrhol sadu príkladov pre testovanie funkčnosti GC. Na základe týchto príkladov som vykonal merania, podľa ktorých som stanovil optimálne parametre GC. Ďalší prínos tejto práce bol v nájdení a popísaní slabých miest implementácií PNtalk a stanovení ďalších možností vývoja.

Dnešný hardvér ponúka vysokú mieru paralelizácie. Tomu sa prispôsobujú aj jednotlivé aplikácie tým, že využívajú viacero vláken a procesov. Tento trend podporuje vznik paralelných verzií algoritmov. Rovnako je to aj v problematike automatickej správy pamäte. Tieto moderné prístupy som sa rozhodol použiť aj v navrhnutom garbage collectore s cieľom využiť čo najviac potenciál hardvéru a čo najmenej spomaliť beh mutátora. To viedlo k použitiu paralelných algoritmov pre *garbage collecting*. Na najvyššej úrovni som využil dvojgeneračný prístup. Ten podľa [1] je veľmi efektívnou alternatívou pre širokú škálu aplikácií.

GC bol overený na sade testov. Z nich vyplýva, že obe implementácie efektívnejsšie pracujú so sieťami metód ako so sieťami objektov. C++ implementácia je limitovaná na cca. 1 000 000 objektov spravovaných pomocou GC. Najväčšie problémy tejto implementácie sú nárast veľkosti objektu prostredia PNtalk, hlavne na 64bitovej architekrúre, a neschopnosť efektívneho uvoľňovania siete objektov. Prvý problém je možné vyriešiť optimalizáciou pridaných atribútov na bitovej úrovni. Druhý problém si vyžaduje hlbšiu zmenu fungovania simulátoru. Dosiahnuté testy ale ukazujú, že pre siete objektov GC zachováva požadovaný výkon a výrazne zlepšuje pamäťové nároky simulácie.

Implementácia GC pre PNtalk v prostredí umožňuje menej optimalizácií na pamäťovej úrovni, ako v prípade C++. Najväčší problem tejto implementácie je v nemožnosti správne detekovať ukončenie simulácie, čo si opäť vyžaduje hlbšiu zmenu implmentácie simulátoru. Opäť je ale možné zaviesť isté pamäťové optimalizácie zlúčením atribútov objektov PNtalk. Na druhej strane implementácia GC v tomto prostredí umožnila dobrý výkon aj pri simuláciách s viac ako 50 000 objektami, pričom pôvodná implementácia nebola schopná pracovať so simuláciou, v ktorej bolo vytvorených cez 1500 objektov.

Literatúra

[1] Johnes, R., Hostings, A., Moss, E.: *The Garbage Collection Handbook: The Art of Automatic Memory Management.* London, CRC Press, 2012, ISBN 978-1-4200-8279-1

[2] Shankar, P., Srikant, Y.N.: *The Compiler Design Handbook: Optimatizations and Machine Code Generation.* London, CRC Press, 2008, ISBN 978-1-4200-4382-2

[3] Janoušek, V.: *Modelování objektů Petriho sítěmi.* FEI VUT v Brně, 1998

[4] Janoušek, V.: *PNtalk* [online]. 14.04.2006 [cit. 10.12.2015]. Dostupný z WWW: <http://perchta.fit.vutbr.cz:8000/projekty/2>

[5] Hanák, M.: *Generování kódu z objektově orientovaných Petriho sítí*, FIT VUT v Brně 2015

[6] Kočí, R.: *PNtalk* [online]. 20.05.2016 [cit. 21.5.2016]. Dostupný z WWW: <http://perchta.fit.vutbr.cz/pntalk2k>

[7] Kočí, R.: *PNtalk C++* [online]. 20.05.2016 [cit. 21.5.2016]. Dostupný z WWW: <http://perchta.fit.vutbr.cz/pntalk2k/2>

Príloha A
Algoritmy garbage collectingu

A.2 Mark-sweep algoritmus

```
collect():
    markFromRoots()
    sweep(HeapStart, HeapEnd)

markFromRoots():
    initialise(worklist)
    for each fld in Roots
        ref ← *fld
        if ref ≠ null && not isMarked(ref)
            setMarked(ref)
            add(worklist, ref)
            mark()

initialise(worklist):
    worklist ← empty

mark():
    while not isEmpty(worklist)
    ref ← remove(worklist)
    for each fld in Pointers(ref)
        child ← *fld
        if child ≠ null && not isMarked(child)
            setMarked(child)
            add(worklist, child)

sweep(start, end):
    scan ← start
    while scan < end
        if isMarked(scan)
            unsetMarked(scan)
        else
            free(scan)
        scan ← nextObject(scan)
```

A.2 Dvojukazovateľový algoritmus

Prebraté z [1].

```
compact():
    relocate(HeapStart, HeapEnd)
    updateReferences(HeapStart, free)

relocate(start, end):
    free ← start
    scan ← end

    while free < scan
        while isMarked(free)
            unsetMarked(free)
            free ← free + size(free) /* find next hole */

        while not isMarked(scan) && scan > free
            scan ← scan - size(scan) /* find previous live object */

        if scan > free
            unsetMarked(scan)
            move(scan, free)
            scan ← free /* leave forwarding adress
                           (destructively) */
            free ← free + size(free)
            scan ← scan - size(scan)

updateReferences(start, end):
    for each fld in Roots /* update roots that pointed to moved
                             objects */
        ref ← *fld
        if ref ≥ end
            *fld ← *ref  /* use the forwarding address left in first
                            pass */
    scan ← start
    while scan < end     /* update fields in live region */
        for each fld in Pointers(scan)
            ref ← *fld
            if ref ≥ end
                *fld ← *ref  /* use the forwarding address left in
                                first pass */
        scan ← scan + size(scan) /* next object */
```

A.3 Kopírovací algoritmus

Prebraté z [1].

```
/* initialisation */
createSemispaces():
    tospace ← HeapStart
    extent ← (HeapEnd - HeapStart) / 2  /* size of a semispace */
    top ← fromspace ← HeapStart + extent
    free ← tospace

atomic allocate(size):
    result ← free
    newfree ← result + size
    if newfree > top
        return null     /* signal `Memory exhausted´ */
    free ← newfree
    return result

atomic collect():
    flip()
    initialise(worklist)                /* empty */
    for each fld in roots               /* copy the roots*/
        process(fld)
    while not isEmpty(worklist)         /* copy transitive closure */
        ref ← remove(worklist)
        scan(ref)

flip():                                 /* switch semispaces */
    fromspace, tospace ← tospace, fromspace
    top ← tospace + extent
    free ← tospace

process(fld):   /* update field with reference to tospace replica */
    fromRef ← *fld
    if fromRef ≠ null
        *fld ← forward(fromRef)  /* update with tospace reference */

forward(fromRef):
    toRef ← forwardingAddress(fromRef)
    if toRef ≠ null    /* not copied (not marked) */
        toRef ← copy(fromRef)
    return toRef

copy(fromRef):    /* copy object and return forwarding address */
    toRef ← free
    free ← free + size(fromRef)
    move(fromRef, toRef)
    forwardingAddress(fromRef) ← toRef    /* mark */
    add(worklist, toRef)
    return toRef
```

A.4 Algoritmus založený na počítaní referencií

Prebraté z [1].

```
New():
    ref ← allocate()
    if ref = null
        error "Out of memory"
    rc(ref) ← 0
    return ref

atomic Write(src, i, ref):
    addReference(ref)
    deleteReference(src[i])
    src[i] ← ref

addReference(ref):
    if ref ≠ null
        rc(ref) ← rc(ref) + 1

deleteReference(ref):
    if ref ≠ null
        rc(ref) ← rc(ref) - 1
        if rc(ref) = 0
            for each fld in Pointers(ref)
                deleteReference(*fld)
            free(ref)
```

A.5 Paralelný mark-sweep algoritmus

Prebraté z [1].

```
shared stealableWorkQueue[N]   /* one per thread */
me ← myThreadId

acquireWork():
    if not is Empty(myMarkStack) /* my mark stack has work to do */
        return
    lock(stealableWorkQueue[me]
    /* grab half of my stealable work queue */
    n ← size(stealableWorkQueue[me]) / 2
    transfer(stealableWorkQueue[me], n, myMarkStack)
    unlock(stealableWorkQueue[me]

    if isEmpty(myMarkStack)
        for each j in Threads
            if not locked(stealableWorkQueue[j])
                if lock(stealableWorkQueue[j])
                    /* grab half of his stealable work queue */
                    n ← size(stealableWorkQueue[me]) / 2
                    transfer(stealableWorkQueue[j], n, my MarkStack)
                    unlock(stealableWorkQueue[j])
                    return

performWork():
    while pop(myMarkStack, ref)
        for each fld in Pointers(ref)
            child ← *fld
            if child ≠ null && not isMarked(child)
                setMarked(child)
                push(myMarkStack, child)


generateWork():    /* transfer all my stack to my stealable work
                      queue */
    if isEmpty(stealableWorkQueue[me])
        n ← size(markStack)
        lock(stealableWorkQueue[me])
        transfer(myMarkStack, n, stealableWorkQueue[me])
        unlock(stealableWorkQueue[me])
```

A.6 Paralelné kopírovanie pomocou zdieľaného zásobníka

Prebraté z [1].

```
shared sharedStack      /* the shared stack of work */
myCopyStack[k]          /* local stack has k slots max.*/
sp ← 0                  /* local stack pointer */

while not terminated()
    enterRoom()         /* enter pop room */
    for i ← 1 to k
        if isLocalStackEmpty()
            acquireWork()
            if isLocalStackEmpty()
                break
        performWork()
    transitionRooms()
    generateWork()
    if exitRoom()       /* leave push room */
        terminate()

acquireWork():
    sharedPop()         /* move work from shared stack */
performWork():
    ref ← localPop()
    scan(ref)           /* see Algorithm 4.2 */
generateWork():
    sharedPush()        /* move work to shared stack */

isLocalStackEmpty()
    return sp = 0

localPush(ref):
    myCopyStack[sp++] ← ref

localPop():
    return myCopyStack[--sp]

sharedPop():            /* move work from shared stack */
    cursor ← FetchAndAdd(&sharedStack, 1) /* try to grab from shared
                                             stack */
    if cursor ≥ stackLimit   /* shared stack empty */
        FetchAndAdd(&sharedStack, -1)     /* readjust stack */
    else
        my CopyStack[sp++] ← cursor[0]    /* move work to local
                                             stack */
shared Push():   /* move work to shared stack */
    cursor ← FetchAndAdd(&sharedStack, -sp) - sp
    for i ← 0 to sp - 1
        cursor[i] ← myCopyStack[i]
    sp ← 0
```

A.9 Paralelný kopírovací algoritmus s dvoma generáciami

```
void copyCollect(vector<GCObject**> roots, unsigned
  whichGeneration):

    /* Výber generácie */
    vector<GCSpacePart> generation =
      chooseGeneration(whichGeneration)

    unsigned toSpaceSize = getTospaceAllocSize()
    toSpace = new GCSpacePart(toSpaceSize)

    queue worklist.clear();

    /* skopírovanie koreňových objektov */
    foreach object in roots:
        process(object, worklist, generation)

    while (not worklist.empty()):
        /* Ak je dosť práce, spusť paralelné kopírovanie */
        if (worklist.size() >= THREADS.size()):
            CopyThread copyThreads[THREADS.size()] =
              createCopyThreads()
            /* rovnomerné počiatočné rozdelenie práce */
            partitiateEqualyQintoPrivateQs(worklist, copyThreads)
            foreach thread in copyThreads:
                thread.start(generation);
            /* Čakajnie na dokončenie kopírovania */
            foreach thread in copyThreads:
                thread.waitForFinish();
            break
        /* potrebná referencia na referenciu, aby sa dalo nastaviť
           nové umiestnenie objektu */
        GCObject** object = worklist.popFront()
        scan(object, worklist, generation)
    freeFromspace(generation)
    /* zmena tospace -> frompsace */
    generation.push(toSpace)
    /* paralelná sweep fáza pre mutexy */
    sweepMutexes()

void scan(GCObject** object, queue worklist, vector<GCSpacePart>
  generation):
    foreach obj in object.getReferences():
        process(object, worklist, generation)
```

```
void process(GCObject** object, queue worklist, vector<GCSpacePart>
  generation)
    GCObject* fromRef = *object
    /* Test na validitu adries, nevalidné adresy sú mimo
       alokovaných generácií, resp. tospace (napr. NULL) */
    if isValidReference(object):
        /* bráni viacnásobnému skopírovaniu */
        fromRef->lock()
        /* zároveň marking fáza pre mutexy, viacnásobné označenie
           tu nie je problém, lebo metóda forward zabráni
           viacnásobnému prechodu grafu objektov */
        fromRef->markMutex()
        /* nastavenie referencie na novú adresu objektu */
        *object = forward(fromRef, worklist, generation)
        fromRef->unlock()

GCObject* forward(GCObject* object, queue worklist,
  vector<GCSpacePart> generation)
    /* Ošetrenie vrátenia správnej adresy pri objektoch z inej
       generácie */
    if (isYoung(generation) && object->isFromOldGen() && ):
        /* Objekt ešte nebol spracovaný, no nesmie sa kopírovať,
        lebo patrí do inej generácie */
        if (not object->isMutexMarked()):
            worklist.push(object)
        GCObject* forwardingAddress = object->getForwardingAddress()
        /* Našiel som objekt pri prechádzaní mladšej generácie,
           ktorý bol ale v tomto cykle presunutý do staršej */
        if forwardingAddress != NULL:
            return forwardingAddress
        /* Objekt staršej generácií, vrátim aktuálne umiestnenie */
        else:
            return fromRef;
    GCObject* forwardingAddress = object->getForwardingAddress()
    /* Ak objekt ešte nebol skopírovaný, kopíruj, inak vráť nové
       umiestnenie v toSpace */
    if forwardingAddress == NULL:
        GCObject* newAddress = copyToTospaceOrOldGen(object)
        object.setForwardingAddress(newAddress)
    return object->getForwardingAddress()

GCObject* copyToTospaceOrOldGen(GCObject* object, queue worklist,
  vector<GCSpacePart> generation):
    object->incSurvivedCycles()
    GCObject* toRef
    if (isYoung(generation) && object.isOldEnought()):
```

```
        toRef = moveToOld(object)
    else:
        toRef = GCAlloc(object->getSize())
        object->cloneTo(toRef)
    object->setForwardingAddress(toRef)
    worklist.push(object)
    return toRef

/* V triede CopyThread */

void start(vector<GCSpacePart> generation):
    /* Kým nie je koniec algoritmu */
    while (true):
        /* Kym nie je fronta prázdna, kvoli synchronizácii pri
           vyvažovaní záťaže riešené takto*/
        while (true):
            GCObject**  object
            try:
                object = privateWorklist.popFront()
            catch (EmptyWorklist):
                break
            scan(object, privateWorklist)
        /* test, ako v algoritme 5.1 */
        worklist.push(checkOtherThreads())
        if privateWorklist.isEmpty():
            break
```

Príloha B
Príklady pre testovanie GC

B.1 TestGC01

```
class TestGC01 is_a PN
object

    place p1(7)
    place p2()
    trans t1
        precond p1(1`x)
        action {
            o1 := TestGC01a new .
            o2 := TestGC01a new .
            o3 := TestGC01a new .
            o4 := TestGC01a new .
            o5 := TestGC01a new .
            r1 := o1 doFor: x .
            r2 := o2 doFor: x .
            r3 := o3 doFor: x .
            r4 := o4 doFor: x .
            r5 := o5 doFor: x .
            r := (((r1 + r2 ) + r3 ) + r4 ) + r5 .
            Transcript show: r1.
            Transcript show: ' '.
            Transcript show: r2.
            Transcript show: ' '.
            Transcript show: r3.
            Transcript show: ' '.
            Transcript show: r4.
            Transcript show: ' '.
            Transcript show: r5.
            Transcript show: ' '.
            Transcript show: r.
            Transcript show: '\n'.
        }
        postcond p2(1`r)

main TestGC01

class TestGC01a is_a PN
object
```

```
method doFor: x
    place x()
    place return()
    trans t1
        precond x(x)
        guard {
            x > 1 .
        }
        action {
            y := x - 1 .
            o1 := TestGC01a new .
            o2 := TestGC01a new .
            o3 := TestGC01a new .
            o4 := TestGC01a new .
            o5 := TestGC01a new .
            o1 doFor: y .
            o2 doFor: y .
            o3 doFor: y .
            o4 doFor: y .
            o5 doFor: y .
        }
        postcond return(y)

    trans t2
        precond x(x)
        guard {
            x <= 1 .
        }
        postcond return(x)
```

B.2 TestGC02

```
class TestGC02 is_a PN
object

    place p1(6)
    place p2()
    trans t1
        precond p1(1`x)
        action {
            o1 := TestGC02a new .
            o2 := TestGC02a new .
            o3 := TestGC02a new .
            o4 := TestGC02a new .
            o5 := TestGC02a new .
            Transcript show: x.
            Transcript show: ' FINISHED\n'.
        }
        postcond p2(1`x)

main TestGC02

class TestGC02a is_a PN
object
    place p1(5)
    place p2()
    trans ta1
        precond p1(1`x)
        action {
            o1 := TestGC02b new .
            o2 := TestGC02b new .
            o3 := TestGC02b new .
            o4 := TestGC02b new .
            o5 := TestGC02b new .
            Transcript show: x.
            Transcript show: ' FINISHED\n'.
        }
        postcond p2(1`x)

class TestGC02b is_a PN
object
    place p1(4)
    place p2()
    trans tb1
        precond p1(1`x)
        action {
            o1 := TestGC02c new .
```

```
            o2 := TestGC02c new .
            o3 := TestGC02c new .
            o4 := TestGC02c new .
            o5 := TestGC02c new .
            Transcript show: x.
            Transcript show: ' FINISHED\n'.
        }
        postcond p2(1`x)

class TestGC02c is_a PN
object
    place p1(3)
    place p2()
    trans tc1
        precond p1(1`x)
        action {
            o1 := TestGC02d new .
            o2 := TestGC02d new .
            o3 := TestGC02d new .
            o4 := TestGC02d new .
            o5 := TestGC02d new .
            Transcript show: x.
            Transcript show: ' FINISHED\n'.
        }
        postcond p2(1`x)

class TestGC02d is_a PN
object
    place p1(2)
    place p2()
    trans td1
        precond p1(1`x)
        action {
            o1 := TestGC02e new .
            o2 := TestGC02e new .
            o3 := TestGC02e new .
            o4 := TestGC02e new .
            o5 := TestGC02e new .
            Transcript show: x.
            Transcript show: ' FINISHED\n'.
        }
        postcond p2(1`x)


class TestGC02e is_a PN
object
    place p1(1)
```

```
place p2()
trans t1
    precond p1(1`x)
    action {

        Transcript show: x.
        Transcript show: ' FINISHED\n'.
    }
    postcond p2(1`x)
```

B.3 TestGC04

```
class TestGC04 is_a PN
object
    place p1(1`5)
    place p2()
    place p3()
    trans t1
        precond p1(1`x)
        action {
            Transcript show: 'T1 ' .
            o1 := TestGC04a new .
        }
        postcond p2(1`o1)
    trans t2
        precond p2(1`o1)
        guard {
            o1 put .
        }
        action {
            Transcript show: 'T2' .
        }
        postcond p3(1`1)

main TestGC04

class TestGC04a is_a PN
object
    place p1(1`1)
    place p2()
    trans t1
        precond p1(1`1)
        action {
            Transcript show: 'T1 ' .
        }
        postcond p2(1`1)
    sync put
        cond p1(1`1)
```

Príloha C
Priložené CD