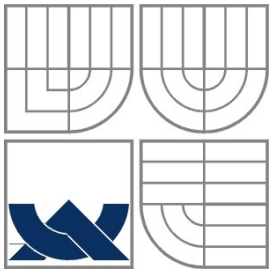
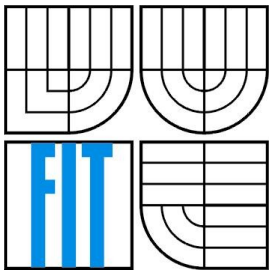


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

DISKRÉTNÍ SIMULACE V JAVĚ: PLÁNOVÁNÍ PROCESŮ

DISCRETE SIMULATION IN JAVA: PROCESS SCHEDULING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

Peter Skočovský

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. David Martinek

BRNO 2009

Abstrakt

Tato práce se zabývá implementací plánování procesů pro diskrétní simulaci v Javě. Rozebírá dva přístupy: implementaci simulačních procesů pomocí vláken a implementaci simulačních procesů bez vláken. Hlavním problémem při použití vláken bylo zabezpečit jejich kooperativní přepínání. Procesy implementované bez vláken jsou rozdělené na atomické části a k přerušení může dojít pouze mezi těmito částmi. Výsledky testování poukazují na to, že implementace pomocí vláken je podstatně pomalejší a paměťově náročnější.

Abstract

This thesis deals with process scheduling implementation for discrete simulation in Java. Two approaches are considered: process implementation using threads and process implementation without threads. Main problem of using threads was to ensure cooperative switching. Processes implemented without threads are divided into atomic parts and suspend can be performed only between these parts. Test results show that implementation using threads is considerably slower and consumes more memory.

Klíčová slova

diskrétní simulace, Java, plánování procesů, vlákno, proces, simulační nástroj

Keywords

discrete simulation, Java, process scheduling, thread, process, simulation tool

Diskrétní simulace v Javě: Plánování procesů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením inžiniera Davida Martinka.
Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Peter Skočovský
20. 5. 2009

Poďakovanie

Rád by som poďakoval vedúcemu mojej práce a mojej rodine, ktorá mi pri písaní práce významne pomohla.

Obsah

1 Úvod

Od nepamäti ľudia skúmajú skutočný svet, čo so sebou prináša rôzne riziká a nepríjemnosti. Často je výhodnejšie skúmať zjednodušený model sveta.

Jednou z možností, ako zjednodušiť pohľad na svet, je vytvorenie jeho diskrétneho modelu. Tento model popisuje deje vo svete pomocou udalostí a procesov, ktoré jeho stav menia skokovo.

Java je v dnešnej dobe často využívaný programovací nástroj. Popularitu získala vďaka tomu, že sa s ňou ľahko a rýchlo pracuje, programy napísané pomocou nej sú prenositeľné medzi rôznymi platformami a obsahuje mnoho štandardných knižníc, ktoré ponúkajú široké možnosti. Preto je vhodné vytvoriť simulačný nástroj v Jave.

Jedným z najdôležitejších problémov, ktoré treba pri vytváraní diskrétneho simulačného nástroja vyriešiť, je plánovanie procesov. Implementácia plánovania procesov ovplyvňuje rýchlosť a pamäťovú efektivitu simulačného nástroja. Preto som sa rozhodol v tejto práci preskúmať problém plánovania procesov.

Nasledujúca kapitola podrobnejšie rozoberá diskrétnu simuláciu. Ďalej je preskúmané, ako je problém plánovania procesov riešený v iných nástrojoch. Následne je vykonaná analýza problému a navrhnuté riešenie. Na konci je popísaná implementáciu riešenia, jej testovanie a zhodnotenie dosiahnutých výsledkov.

2 Diskrétna simulácia

V tejto kapitole sa budem zaoberať diskrétnou simuláciou a zhrniem jej vlastnosti, ktoré sú dôležité pre túto prácu.

2.1 Modelovanie a simulácia

Pri experimentovaní s reálnymi systémami často narážame na rôzne problémy, ako sú časová a finančná náročnosť, alebo bezpečnostné riziká. Preto je v niektorých prípadoch výhodné vytvoriť model systému a experimentovať s ním.

Model systému je systém, ktorý sa chová podobne, ako jeho vzor. Model má homomorfný vzťah ku vzoru. Inými slovami povedané model je vzor zjednodušený tak, aby napodobňoval najmä tie vlastnosti vzoru, ktoré sú predmetom skúmania [1]. Napríklad pri modelovaní obsluhy požiadavky pre webový server nás nezaujímajú všetky vlastnosti servera, ani to, čo požiadavka vlastne obsahuje. Zaujíma nás iba to, kedy požiadavka príde, a ako dlho bude trvať obsluha.

Existuje zobrazenie, ktoré mapuje vzor na model a pritom zachováva podstatné vlastnosti vzoru [1]. Napríklad v predchádzajúcom príklade by toto zobrazenie mohlo priraďovať trvaniu obsluhy určitú číselnú hodnotu. Takéto zobrazenie sa nazýva modelovanie. Toto zobrazenie je nejednoznačné, to znamená, že jeden systém môžeme modelovať viacerými modelmi podľa účelu a úrovne abstrakcie a jeden model môže zodpovedať viacerým vzorom [1]. Napríklad doba obsluhy nemusí byť modelovaná číselnými hodnotami, ale napr. dobou, za ktorú som schopný odpovedať na otázku, ktorú sa ma niekto spýtal. A naopak tie isté číselné hodnoty modelu môžu modelovať obsluhu akejkoľvek požiadavky v akomkoľvek systéme, napríklad vybavenie požiadavky serverom, alebo vybavenie zákazníka pri pokladni.

Simulácia je proces experimentovania s modelom. Produktom simulácie sú určité výsledky. Ak sa tieto výsledky zobrazia zobrazením inverzným k zobrazeniu, pomocou ktorého vznikol model, získame vedomosti o vzore. Napríklad výsledkom experimentovania s modelom z predchádzajúcich príkladov budú určité čísla. Ak tieto čísla zobrazíme zobrazením inverzným k tomu, pomocou ktorého sme z doby obsluhy získali čísla (k zobrazeniu, ktoré sme použili na modelovanie), získame údaje o dobe trvania procesov v modelovanom systéme.

Pri zobrazení reálneho systému na jeho model (modelovaní) sa transformuje aj čas, v ktorom prebiehajú deje reálneho systému. Preto počas simulácie nebude čas pre model ubiehať tak ako reálny čas. Takýto čas pre model sa nazýva *simulačný čas*. Napríklad pri simulovaní doby obsluhy

požiadavky ubehne toľko simulačného času, koľko trvá táto obsluha. Reálneho času však môže uplynúť rôzne množstvo, ktoré bude závisieť napríklad od toho, ako dlho trvá prepísať hodnotu premennej, v ktorej je uložený simulačný čas (prípadne iná réžia).

Výhody modelovania a simulácie sú:

- Flexibilita experimentov
  - Experimenty sú časovo flexibilné. Na rozdiel od experimentovania s reálnym systémom je možné ich trvanie a načasovanie prispôsobiť podľa potreby.
  - Výsledky je možné odoberať v akomkoľvek okamihu simulácie.
  - Je možné opakovať rovnaký experiment viackrát s rovnakými alebo s rôznymi parametrami.
- Experimenty na modeloch sú často finančne výhodnejšie ako experimenty s reálnymi systémami.
- Experimenty s reálnymi systémami môžu byť nebezpečné.
- Niekedy je iný spôsob získavania vedomostí o systéme príliš náročný, alebo dokonca až nemožný.

Nevýhody modelovania a simulácie sú:

- Kvôli zjednodušovaniu pri vytváraní modelu môžu byť výsledky skreslené a nepresné.
- Simulácia často vyžaduje náročné výpočty.
- Výsledkom experimentov sú konkrétne hodnoty a pri zmene parametrov modelu treba vykonať celú simuláciu znova.

Modelovať reálne systémy je možné pomocou akýchkoľvek iných vhodných systémov. Táto práca sa ale obmedzí na modelovanie a simuláciu pomocou počítačového programu.

2.2 Diskrétna simulácia

Diskrétna simulácia je druh simulácie, pri ktorej sa stav modelu mení diskrétne. Nezáleží na tom, ako sa mení simulačný čas, v určitých momentoch sa stav modelu zmení skokovo. Tieto zmeny majú v simulačnom čase nulové trvanie. Takéto momenty sa nazývajú *udalosti*.

Vlastnosti udalostí:

- Udalosti sú atomické. Celé sa vykonajú v jednom momente simulácie a nie je možné ich prerušiť.
- Počas simulácie sa udalosti môžu dynamicky plánovať. To znamená, že je možné pre udalosti nastaviť čas, kedy sa majú vykonať, alebo je možné tento čas vykonania udalostí zmeniť (preplánovať).

Dobou medzi udalosťami sa nemá zmysel zaoberať, lebo sa počas nej stav modelu nemení. Udalosť často vytvára, plánuje, alebo ruší iné udalosti.

Z vyššie popísaných faktov vyplýva princíp „*discrete event*“ simulácie [1]. Naplánované udalosti sú uložené v štruktúre zvanej kalendár (scheduler, future event list). Sú zoradené podľa času, v ktorom majú nastať.

Algoritmus riadenia diskrétnej simulácie sa nazýva „*next event*“ algoritmus. Prebieha nasledovne:

- Kým kalendár nie je prázdny, alebo nevypršal simulačný čas:
  - Z kalendára sa vyberie udalosť naplánovaná na najbližší čas.
  - Simulačný čas sa posunie na čas, kedy sa má táto udalosť vykonať.
  - Udalosť sa vykoná.

V reálnych systémoch väčšinou prebieha súbežne viacero dejov. Spôsob, ako deje menia stav systému, sa dá namodelovať pomocou udalostí a trvanie dejov je možné namodelovať časovými intervalmi medzi týmito udalosťami. Udalosti modelujúce jeden dej spolu súvisia, logicky za sebou nasledujú a navzájom sa podmieňujú a ovplyvňujú. Preto je výhodnejšie a jednoduchšie modelovať deje v modelovaných systémoch pomocou *simulačných procesov*, ktoré združujú navzájom súvisiace udalosti.

Simulačné procesy musia byť schopné prebiehať súbežne, rovnako ako deje v reálnych systémoch, ktoré modelujú. To pri behu simulácie väčšinou ale nie je možné. Diskrétna simulácia sa obvykle vykonáva na platformách, kde je možné v jednom okamihu reálneho času vykonávať iba jednu udalosť. To ale nebráni tomu, aby sa vykonávalo viac udalostí v jednom okamihu simulačného času, a už vôbec nie tomu, aby sa z pohľadu modelu zdalo, že simulačné procesy bežia paralelne. Vďaka tomuto faktu je možné dosiahnuť *kvaziparalelizmus*. Aby bolo možné simulačné procesy vykonávať kvaziparalelne, musí byť možné ich vykonávanie pozastaviť a neskôr znova obnoviť. Kvaziparalelizmus sa potom dosahuje tak, že simulačné procesy sa počas svojho vykonávania prerušujú, čím umožnia beh iných procesov a neskôr, keď sa tieto iné procesy ukončia alebo prerušia, vrátia sa pôvodné procesy k svojmu vykonávaniu znova. Keď sa takto vo svojom vykonávaní strieda viacero procesov, javia sa, akoby bežali paralelne.

Z toho vyplývajú nasledujúce vlastnosti simulačných procesov:

- Nie sú atomické. Nemusia sa vykonať v jednom časovom momente simulácie. Keď sa raz vykonávanie začne, môže sa pozastaviť a neskôr obnoviť.
- Podobne ako udalosti, aj procesy sa môžu počas simulácie dynamicky plánovať. Pre procesy je možné nastaviť čas, kedy sa majú vykonať alebo v ich vykonávaní pokračovať, alebo tento čas upraviť.

Takže jediný zásadný rozdiel medzi udalosťami a procesmi je v tom, že udalosti sú atomické a procesy nie.

Preto jedným z najväčších problémov, ktorý treba pri vytváraní nástroja na diskrétnu simuláciu vyriešiť, je implementácia simulačných procesov.

3 Podobné projekty

V tejto kapitole sa budem zaoberať tým, ako je problém plánovania procesov riešený v iných simulačných nástrojoch. Najprv rozoberiem simulačnú knižnicu SIMLIB/C++ a potom niekoľko simulačných nástrojov vyvinutých v Jave.

3.1 SIMLIB/C++

SIMLIB/C++ je voľne dostupná simulačná knižnica pre programovací jazyk C++ s otvoreným zdrojovým kódom. Aktuálna verzia umožňuje popis spojitých, diskrétnych, kombinovaných, 2D a 3D vektorových, ako aj fuzzy modelov. Knižnica je vyvinutá na Fakulte Informačných Technológií, Vysokého Učení Technického v Brne. Vývoj začal v roku 1991. Zdrojové kódy sú dostupné a fungujú pod operačnými systémami Linux, FreeBSD a MS Windows. SIMLIB/C++ zvláda až 70 miliónov bežiacich simulačných procesov na počítači so 64-bitovým procesorom a 32GB pamäte (približne 500 bytov na proces). [2]

V knižnici SIMLIB/C++ sa deje prebiehajúce v modelovanom systéme modelujú pomocou programu, ktorý je zapísaný priamo v programovacom jazyku C++. Tento program je vložený do funkcie `Behavior()` triedy `Process`. (V triede `Event` sa zmena stavu popisuje rovnakým spôsobom.) Kontexty procesov sa prepínajú pomocou kopírovania programového zásobníka. Pri prepnutí procesov sa uloží aktuálny obsah programového zásobníka a načíta sa pôvodný obsah. Kontext procesora sa mení pomocou funkcií `setjmp()` a `longjmp()`. Viac informácií v [3].

3.2 Simkit

Simkit je knižnica na vytváranie modelov pre „Discrete Event“ simuláciu napísaná v Java 2. Bola vyvinutá na Naval Postgraduate School v Monterey v Kalifornii. Knižnica Simkit podporuje vytváranie „Event Graph“ modelu. Základné elementy „Event Graph“ modelu sú mapované priamo na bloky kódu, ktoré využívajú knižnicu Simkit. Udalosti sú v knižnici Simkit reprezentované metódami. Na ich plánovanie sa používa reflexia. Knižnica Simkit nepodporuje simulačné procesy. [4]

3.3 DESMO-J

DESMO-J je knižnica určená na diskrétne modelovanie a simuláciu napísaná v Jave. Bola vyvinutá na University of Hamburg, Department of Computer Science. Knižnica DESMO-J obsahuje zvláštnu triedu pre popis modelu a zvláštnu triedu pre popis experimentu. To umožňuje vykonať rovnaký experiment na rôznych modeloch (ktoré môžu reprezentovať napríklad alternatívne návrhy systému), rovnako ako vykonať rôzne experimenty na tom istom modeli. Knižnica DESMO-J podporuje simulačné procesy, na ktorých implementáciu využíva vlákna. Ďalej obsahuje rôzne generátory náhodných čísel, umožňuje výstup v HTML a XML, umožňuje zber štatistík atď. Autori knižnice plánujú vytvoriť simulačné prostredie založené na platforme Eclipse. [5]

3.4 simjava

simjava je knižnica založená na procesoch určená na „discrete event“ simuláciu pre Javu. Bola vyvinutá na University of Edinburgh, Institute for Computing Systems Architecture. Simulácia v simjava je kolekcia entít, z ktorých každá beží vo svojom vlastnom vlákne. Tieto entity sú navzájom pospájané portami a môžu komunikovať prostredníctvom zasielaní udalostí. Trieda centrálneho systému riadi všetky vlákna, posúva simulačný čas a doručuje udalosti. Priebeh simulácie je zaznamenávaný pomocou správ, ktoré produkujú entity, a môže byť uložený do súboru. Entity a porty sú identifikované textovými reťazcami. Chovanie entity je definované jednou z jej metód. Priebeh simulácie je možné vizualizovať. [6]

3.5 J-Sim

J-Sim je kompozičné simulačné prostredie založené na komponentoch. Bolo vyvinuté na Ohio State University, Department of Electrical & Computer Engineering. J-Sim je napísané v Jave a postavené na architektúre autonómnych komponentov, ktorá je podobná COM/COM+, JavaBeansTM, alebo CORBA. Každý komponent pracuje samostatne (vo vlastnom vlákne) a s ostatnými komunikuje pomocou portov, čím sa podobá na predchádzajúci nástroj. Riadenie simulácie stráži hornú hranicu množstva súčasne spustených vláken. Keby sa mala táto hranica prekročiť, počká, kým sa niektoré z bežiacich vláken neukončia. [7]

3.6 JiST

JiST je vysoko výkonný diskrétny simulačný nástroj, ktorý beží nad štandardnou Java Virtual Machine. Bol vyvinutý na Cornell University. Tento simulačný nástroj je prekvapujúco efektívny. Je časovo aj pamäťovo efektívnejší ako niektoré existujúce vysoko optimalizované simulačné nástroje. Napríklad zvládne vykonať približne dvakrát toľko udalostí za jednotkový čas ako vysoko optimalizovaný simulačný nástroj Parsec, ktorý je napísaný v programovacom jazyku C, a procesne orientovanú simuláciu zvládne s využitím zlomku pamäte. Takýto výkon JiST dosahuje vďaka tomu, že pred spustením simulácie prepíše súbory, v ktorých je skompilovaný simulačný program. Simulačný program pre JiST je zapísaný v Jave a skompilovaný štandardným kompilátorom jazyka Java. Skompilované triedy sú potom modifikované prepisovačom tak, aby mohli byť vykonávané na simulačnom jadre. Vďaka tomu JiST nemusí používať vlákna na to, aby podporoval simulačné procesy. Simulačný program, prepisovač a simulačné jadro sú napísané v jazyku Java. [8]

4 Analýza

V tejto kapitole rozanalyzujem problémy spojené s vývojom diskrétneho simulačného nástroja. Najprv špecifikujem ciele a požiadavky na vyvíjaný simulačný nástroj a následne rozoberiem možnosti implementácie prepínania procesov.

4.1 Ciele

Cieľom tejto práce je vyvinúť diskrétny simulačný nástroj s podporou simulačných procesov. Nástroj bude vykonávať „discrete event“ simuláciu tak, ako je popísaná v kapitole 2.2 a bude podporovať abstrakciu simulačných procesov.

4.2 Požiadavky

Vyvíjaný diskrétny simulačný nástroj musí spĺňať nasledujúce požiadavky:

- Nástroj musí byť implementovaný v Jave.
- Simulácie bývajú obvykle rozsiahle a časovo náročné. Preto simulačný nástroj musí byť časovo a pamäťovo efektívny.
- V simulačnom nástroji musí byť možné namodelovať akýkoľvek dej prebiehajúci v modelovanom systéme. Inými slovami povedané, musí mať dostatočne veľkú vyjadrovaciu silu na to, aby bol schopný vyjadriť akýkoľvek algoritmus.
- Nástroj by mal byť ľahko a rýchlo použiteľný a nemal by byť príliš zložitý, teda aby bol používateľsky príjemný.

4.3 Procesy

Jadro „discrete event“ simulátora je jednoduché. Jednoducho iba vykonáva „next event“ algoritmus, popisuje abstrakciu udalosti, prípadne ponúka inú podporu. Preto jedinou vecou, ktorou sa treba vážne zaoberať, je implementácia simulačných procesov. Koniec-koncov, to je úlohou tejto práce.

Simulačný proces musí byť plánovateľný, musí byť schopný bežať paralelne s ostatnými procesmi a počas svojho behu musí vykonávať algoritmus, ktorý diskrétne modeluje vybraný dej v modelovanom systéme.

Tieto podmienky by sa dali splniť aj bez abstrakcie procesu. Deje prebiehajúce v modelovanom systéme sa dajú modelovať iba pomocou udalostí. A to tak, že každá udalosť zaistí naplánovanie udalosti, ktorá za ňou v modelovanom deji logicky nasleduje. Takýto prístup využíval napríklad simulačný nástroj popísaný v kapitole 3.2. Toto je klasický „discrete event“ prístup, preto má plnú vyjadrovaciu silu a nezníženú efektivitu. Lenže modelovanie dejov iba pomocou udalostí je zložitejšie ako pomocou procesov. Model obsahuje oveľa viac udalostí, treba si pamätať, ktoré navzájom súvisia a nesmie sa na žiadnu zabudnúť. Zmenšiť tieto nevýhody by mohlo používanie určitých návrhových postupov alebo formalizmov, podobne, ako to realizuje simulačný nástroj popísaný v kapitole 3.2.

Na druhú stranu silná abstrakcia procesu by vyzerala tak, že jeden dej prebiehajúci v modelovanom systéme by bol modelovaný jedným simulačným procesom. Tento proces by musel byť schopný pozastaviť svoje vykonávanie a zaistiť naplánovanie svojho obnovenia. Efektivita a vyjadrovacia sila takéhoto prístupu by bola otázna a silne by závisela od jeho implementácie. Tento prístup by bol ale rozhodne používateľsky príjemnejší, lebo jeden dej je modelovaný jedným procesom a nie množstvom udalostí.

Preto je potrebné vyriešiť, ako implementovať algoritmus modelujúci dej prebiehajúci v modelovanom systéme tak, aby bolo možné ho pozastaviť a znova obnoviť.

Jednou z možností je použiť vlákna. Java implicitne ponúka podporu vláken. (Táto skutočnosť bude bližšie popísaná v kapitole 5.1). Algoritmus modelujúci dej by bol jednoducho zapísaný v jednej funkcii. Táto funkcia by bola spustená vo vlastnom vlákne, a teda by ju bolo možné v ktoromkoľvek momente pozastaviť a neskôr obnoviť. Okrem toho vlákna ponúkajú aj riešenie paralelizmu, ktorý ale bude simulátor riešiť inak. Preto je riešenie pomocou vláken príliš všeobecné pre účely tejto práce. Rovnako prepínanie vláken bude trvať nejakú dobu a každé vlákno bude mať určitú spotrebu pamäti. Preto implementácia pomocou vláken môže byť menej efektívna.

Ďalšou možnosťou je rozdeliť algoritmus modelujúci dej na niekoľko častí (*operácií*), medzi ktorými by sa dal pozastaviť. Každá operácia by bola popísaná vlastnou funkciou. Každá operácia by sa vykonala vcelku, nedala by sa prerušiť (bola by atomická). Proces by sa dal pozastaviť iba medzi týmito operáciami. Preto, aby sa pozastavený proces dokázal znova obnoviť v správnom mieste, musel by si pamätať, kde bol pozastavený, alebo ktorá operácia práve nasleduje.

Pri implementácii tohto prístupu si treba dať pozor na to, aby mal plnú vyjadrovaciu silu. Tá záleží na tom, ako sú operácie usporiadané. Keby napríklad jednoducho nasledovali v jednom statickom poradí za sebou, určite by sa pomocou nich nedal zapísať akýkoľvek algoritmus. (Na to, aby sa dal zapísať akýkoľvek algoritmus, sú potrebné príkazy vetvenia a cyklu.) Tento problém by sa dal

vyriešiť napríklad tak, že každá operácia by určila, ktorá operácia nasleduje za ňou, alebo tak, že by boli umožnené skoky v zozname operácií.

Tento prístup nie je oveľa menej efektívny ako simulovanie iba pomocou udalostí. Vlastne je to veľmi podobný prístup. Operácie by sa dali chápať ako udalosti. Rozdiel je iba v tom, že existuje abstrakcia procesu, ktorý združuje udalosti modelujúce jeden dej. Vďaka tejto abstrakcii procesu je zápis modelu jednoduchší a prehľadnejší ako modelovanie iba pomocou udalostí, ale zložitejší ako prístup pomocou vláken. Je jasné, ktoré udalosti (v tomto prípade operácie) patria ku ktorému procesu a ako za sebou nasledujú. Treba si ale stále dávať pozor na to, ako je proces rozdelený na operácie. Používateľská príjemnosť by sa ešte dala zvýšiť tak, že modely by boli zapisované v externom jazyku a následne kompilované do takejto vnútornej reprezentácie. To by ale ochudobnilo simulačný nástroj o veľa výhod, ktoré ponúka Java. Keby sme chceli tieto výhody zanechať, bolo by možné zapísať algoritmus popisujúci dej v simulačnom systéme do jednej funkcie. (Tak, ako pri prístupe s využitím vláken.) Po skompilovaní súboru s touto funkciou do skompilovanej triedy by sa ale táto trieda musela prepísať tak, aby sa funkcia rozdelila na operácie a vznikla by štruktúra potrebná pre podporu procesu. Existuje štandardná knižnica pre Javu, ktorá takýto prepis umožňuje. Takýto prístup využíva simulačný nástroj popísaný v kapitole 3.6.

5 Dôležité fakty o Jave

Vzhľadom na to, že vyvíjaný nástroj bude implementovaný v Jave, je nutné rozobrať niekoľko jej vlastností, ktoré budú pre prácu kľúčové.

5.1 Vlákna

Java implicitne podporuje vlákna. Pri každom spustení Java Virtual Machine ich beží viacero (minimálne to, v ktorom beží hlavný program). Každé vlákno má vlastný programový zásobník, ku ktorému má prístup iba ono samo. Do zásobníka sa ukladajú iba premenné jednoduchých typov a odkazy na objekty. Samotné objekty sú uložené v pamäťovom priestore nazvanom *heap* (v preklade hromada). Na heap sa ukladajú vždy iba objekty. K dátam na heap majú prístup všetky vlákna. [9] Preto ak prístup vláken k dátam na heap nebude synchronizovaný, môžu sa tieto dáta dostať do nekonzistentného stavu. Java podporuje dva prístupy k synchronizácii vláken: vzájomné vylučovanie a kooperáciu vláken. Vzájomné vylučovanie umožňuje viacerým vláknam nezávisle pracovať v zdieľanej pamäti bez toho, aby si navzájom do tejto práce zasahovali. Kooperácia umožňuje vláknam spolupracovať pri dosahovaní spoločného cieľa. [9] Na účely synchronizácie vláken sa využívajú zámky a monitory.

5.2 Zámky a monitory

Každý objekt je spätý s jednou zámkou. Vlákna môžu zámky zamykať alebo odomykať. Jednu zámku môže mať naraz zamknuté iba jedno vlákno.

Monitor stráži časť programu a zaisťuje, že ho bude vykonávať v jednom čase iba jedno vlákno. Na to používa zámky. Každý monitor je spätý s odkazom na objekt. Ak chce vlákno vykonať časť programu, ktorá je strážená monitorom (vstúpiť do monitora), musí zamknúť zámku objektu, na ktorý sa monitor odkazuje. Ak je táto zámka už zamknutá, musí vlákno čakať, kým sa zámka znova neodomkne (čakať na entry set monitora). Keď sa zámka odomkne, a na jej odomknutie čakalo viacero vláken, musia súťažiť o to, ktoré si zámku môže zamknúť teraz. Keď vlákno dokončí stráženú časť programu (opúšťa monitor), musí zámku znova odomknúť. [9] Takto monitor podporuje vzájomné vylučovanie vláken.

Okrem toho monitor zaisťuje aj kooperatívnu synchronizáciu vláken. Ak vlákno vstúpilo do monitora, môže ostať čakať na jeho wait set a nechať bežať iné vlákno. Bežiace vlákno môže zasa

prebudiť vlákna čakajúce na wait set monitora. To sa dá vykonať pomocou funkcií `wait()` a `notifyAll()`, ktoré sú popísane v nasledujúcej podkapitole.

5.3 Metódy `wait()` a `notifyAll()`

Vlákno sa dá pozastaviť pomocou metódy `wait()`. Zavolanie tejto metódy nad určitým objektom spôsobí, že aktuálne vlákno ostane čakať na wait set monitora spätého s objektom, nad ktorým je táto metóda volaná. Na to, aby mohlo vlákno vstúpiť do wait set monitora, musí najprv vstúpiť do tohto monitora. Z toho vyplýva, že pri volaní metódy `wait()` musí volajúce vlákno vlastniť zámku objektu, nad ktorým túto metódu volá, v opačnom prípade bude vyvolaná výnimka `IllegalMonitorStateException`. Po zavolaní metódy `wait()` vlákno uvoľní zámku daného objektu a zaspí. Po prebudení vlákno najprv počká, kým bude môcť znova získať zámku daného objektu, keď ju získa, znova pokračuje vo svojom behu. [10]

Metódou `notifyAll()` nad určitým objektom je možné prebudiť všetky vlákna, ktoré čakajú na wait set monitora spätého s objektom, nad ktorým je táto metóda volaná. Pri volaní metódy `notifyAll()` musí volajúce vlákno vlastniť zámku objektu, nad ktorým túto metódu volá, v opačnom prípade bude vyvolaná výnimka `IllegalMonitorStateException`. Preto keď nejaké vlákno zavolá metódu `notifyAll()` nad určitým objektom a prebudí iné vlákna, tieto vlákna budú môcť pokračovať vo svojom behu, až keď prebúdzajúce vlákno uvoľní zámku daného objektu. [10]

Existuje aj metóda `notify()`. Tá ale prebúdza iba jedno ľubovoľné čakajúce vlákno; ktoré to je, závisí od implementácie. [10]

6 Návrh

V tejto kapitole sa budem zaoberať návrhom jednoduchého simulačného nástroja napísaného v Jave. Najprv rozoberiem návrh jadra simulátora a potom sa bližšie zameriam na návrh simulačných procesov.

6.1 Jadro simulátora

V tejto podkapitole je popísaný návrh jednoduchého jadra diskrétneho simulátora. Návrh je inšpirovaný simulačnou knižnicou SIMLIB/C++.

Jadro sa bude skladať z týchto funkčných celkov:

- Riadenie simulácie

Riadenie simulácie pozostáva z inicializácie, behu, prípadne ukončenia simulácie. Riadenie simulácie bude spravovať simulačný čas a kalendár. Požadovanou vlastnosťou simulátora je, aby pracoval s viacerými simuláciami, pričom každá z nich bude potrebovať vlastné riadenie. Špecifikovať pri každom vytváraní objektu, do ktorej simulácie patrí, by bolo zdĺhavé, preto bude najlepšie, ak bude existovať trieda pre riadenie simulácie a v simulátore bude jej implicitná inštancia, ktorá sa bude implicitne priraďovať vytváraným objektom, ak nebude špecifikované inak.

- Udalosti

Udalosti bude možné plánovať (vkladať do kalendára) a vykonávať (budú obsahovať metódu, ktorá bude definovať ich chovanie).

- Simulačné procesy

Budú popísané vo zvláštnej podkapitole.

- Kalendár a fronty simulačných procesov

Aj kalendár, aj fronty simulačných procesov sú v podstate prioritné fronty. Rozdiel medzi nimi je iba v tom, že obsahujú iné elementy, ktoré zoraďujú podľa iného kľúča. Funkčnosť majú rovnakú. Preto kalendár i fronty procesov budú mať jedno generické rozhranie, ktoré sa bude používať v celom simulátore. Kvôli tomu pravdepodobne klesne časová efektivita, ale pre obe bude stačiť jedna implementácia.

- Spracovávanie požiadaviek

Triedy pre zariadenia a sklady budú mať rovnakú funkčnosť ako v SIMLIB/C++.

- Zber štatistík

Rovnako ako v SIMLIB/C++.

- Generovanie náhodných čísel

Niekoľko funkcií pre generovanie náhodných čísel podľa rôznych rozložení bude v triede riadenia simulácie.

6.2 Simulačné procesy

Rozhodol som sa implementovať a otestovať dva rôzne prístupy: implementáciu procesov pomocou vláken a implementáciu procesov bez vláken. Každý prístup bude implementovaný vo vlastnej triede. Pretože majú oba rovnakú funkčnosť, ktorú odlišne implementujú, obe triedy budú dediť od abstraktnej triedy procesu, ktorá sa bude používať v celom simulátore. Trieda abstraktného procesu bude implementovať funkčnosť potrebnú pre plánovanie, pretože táto funkčnosť je pre oba varianty rovnaká. Zdedené triedy budú implementovať funkčnosť pre spustenie, pozastavenie a ukončenie vykonávania procesu.

6.2.1 Implementácia procesov pomocou vláken

Algoritmus modelujúci dej v modelovanom systéme bude popísaný v určitej metóde. Táto metóda bude bežať vo vlastnom vlákne. To, kedy bude bežať ktoré vlákno, bude riadiť riadenie simulácie pri plánovaní procesov. Preto treba vyriešiť iba to, ako prepínať vlákno riadenia simulácie a vlákno jedného simulačného procesu. Na tento účel sa použije kooperatívna synchronizácia vláken, čo znamená, že vlákna sa budú uspávať a navzájom prebúdzať pomocou monitorov. Na jednom monitore bude čakať vlákno riadenia simulácie, a na druhom vlákno simulačného procesu. Pri spustení procesu sa vytvorí a spustí vlákno tohto procesu a uspí sa vlákno, v ktorom beží riadenie simulácie. Pri pozastavení procesu sa uspí vlákno tohto procesu a znova spustí vlákno riadenia simulácie. Pri opätovnom spustení procesu sa prebudí vlákno tohto procesu a uspí vlákno riadenia simulácie. Keď proces ukončí svoje vykonávanie, znova prebudí vlákno riadenia simulácie.

Takže pri prepínaní vláken sa vždy jedno vlákno uspí a druhé sa prebudiť. Vlákno *t1*, ktoré má zaspať, musí najprv prebudiť vlákno *t2*, ktoré má pokračovať vo vykonávaní. V opačnom prípade by sa program zablokoval (obe vlákna by spali). V čase medzi prebudením vlákna *t2* a uspaním vlákna *t1* sa môže stať, že sa vlákno *t2* pokúsi prehodiť vlákna opačným smerom. To znamená, že by sa najprv pokúsilo prebudiť vlákno *t1*, (ktoré ale ešte nespalo, takže by sa nič nestalo) a potom by zaspalo samotné vlákno *t2*. Potom by sa znova aktivovalo vlákno *t1*, ktoré by (podľa plánu) zaspalo.

Znova by sa program zablokoval, pretože by obe vlákna spali. Preto treba zaručiť, aby sa vlákno *t2* nepokúsilo prebudiť vlákno *t1*, kým to ešte nespí.

Na to je možné využiť monitory, na ktorých budú vlákna čakať. Vlákno *t1* vždy najprv vojde do monitora, na ktorom bude čakať. Tým zamkne zámku, ktorú bude potrebovať vlákno *t2*, aby prebudilo vlákno *t1*. Až potom vlákno *t1* vojde do monitora, na ktorom čaká vlákno *t2* (pričom musí zamknúť zámku tohto monitora), prebudí vlákno t2 a tento monitor opustí. Potom zaspí vlákno *t1* v monitore, do ktorého už vošlo na začiatku.

6.2.2 Implementácia procesov bez použitia vláken

Algoritmus modelujúci dej v modelovanom systéme bude rozdelený do atomických operácií. Každá operácia bude definovaná jednou metódou. Proces sa nebude dať pozastaviť počas vykonávania operácie, iba medzi vykonávaním operácií. Operácie budú uložené v poli a pri behu procesu sa budú vykonávať za sebou. Pre dosiahnutie celej vyjadrovacej sily bude mať trieda tohto procesu chránenú metódu, pomocou ktorej sa bude dať určiť, ktorá operácia sa má vykonať ako nasledujúca. Pri spustení procesu sa začnú operácie vykonávať zaradom. Pri pozastavení procesu sa dokončí práve vykonávaná operácia, proces si zapamätá, ktorá operácia má nasledovať a vo vykonávaní programu bude pokračovať riadenie simulácie. Pri opätovnom spustení procesu sa pokračuje vo vykonávaní operácií od tej, ktorú si proces zapamätal. Keď sa vykonajú všetky operácie, vo vykonávaní programu bude opäť pokračovať riadenie simulácie.

7 Implementácia

V tejto kapitole popisujem implementáciu jednoduchého simulačného nástroja. Simulačný nástroj je implementovaný ako knižnica pre Javu. Táto knižnica je umiestnená v balíčku `simlib_j`.

7.1 Riadenie simulácie

Riadenie simulácie zabezpečuje trieda `Simulation`. Táto trieda spravuje simulačný čas a kalendár. Simulačný čas má typ `double` a je možné ho zistiť pomocou metódy `getTime()`. Kalendár je viditeľný iba pre triedy v balíčku knižnice. Používateľ k nemu teda nemá priamy prístup. Trieda `Simulation` ďalej obsahuje algoritmus pre riadenie simulácie. Simulácia sa inicializuje metódou `init()`, pomocou ktorej je možné zadať časové obmedzenie simulácie. Simulácia sa spúšťa pomocou metódy `run()`, ktorá obsahuje spomínaný algoritmus riadenia simulácie. Simuláciu je možné ukončiť metódou `stop()`. V opačnom prípade sa simulácia ukončí po vypršaní určeného času, alebo keď už nebude naplánovaná žiadna udalosť. Je možné, že po konci simulácie ostanú niektoré procesy nedokončené. V prípade implementácie procesov pomocou vláken ostanú tieto vlákna uspané. Trieda `Simulation` zaisťuje ich ukončenie. Na začiatku simulácie je potrebné niektoré objekty inicializovať (vyčistiť) (`Queue`, `Facility`, `Store` …). Aj toto zabezpečuje trieda `Simulation`.

7.2 Kalendár a fronty procesov

Ako je spomenuté už v návrhu, aj kalendár, aj fronty procesov sú prioritné fronty. Preto jadro simulátora na tieto účely používa abstraktnú triedu `PriorityQueue`. `PriorityQueue` je generická trieda. To znamená, že ako priorita sa dá použiť akákoľvek trieda, ktorá je porovnateľná sama so sebou (implementuje rozhranie `Comparable`) a elementy môžu byť ľubovoľné. Trieda `PriorityQueue` obsahuje statické „factory“ metódy, ktoré vracajú jej inštanciu, takže implementáciu tejto triedy je možné neskôr jednoducho nahradiť. Zatiaľ je vytvorená jedna implementácia (`TreeMapQueue`), ktorá ale nie je veľmi efektívna.

Poznámka: Najefektívnejšie by bolo, keby boli fronty procesov a kalendár vytvorené podobne ako v SIMLIB/C++. Každý element môže byť naraz iba v jednej fronte, preto dáta potrebné pre vloženie elementu do fronty (priorita, ukazovatele na susedov …) môžu byť súčasťou elementu. Vďaka tomu by sa uvedené dáta vytvárali iba raz a bolo by ich možné použiť znovu. Otázkou je, či sa

môže stať, že element bude zároveň vo fronte, aj v kalendári. Ak áno, kalendár by bolo treba riešiť zvlášť.

7.3 Udalosti

Udalosti sú implementované v triede `Event`. Majú rozhranie, ktoré ich umožňuje plánovať (vkladať do a vyberať z kalendára). Zmenu stavu modelu popisuje metóda `behavior()`, ktorá je abstraktná. Ak chce používateľ triedu `Event` použiť, musí vytvoriť novú triedu, ktorá bude od triedy `Event` dediť, a metódu `behavior()` predefinovať.

7.4 Procesy

Aby bolo možné vytvoriť dve rôzne implementácie procesov a jednoducho ich zamieňať, je implementovaná abstraktná trieda `AbstractProcess`. Celé jadro pracuje iba s touto triedou. Od nej dedia triedy `ThreadProcess` a `OperationProcess`, ktoré implementujú konkrétnu funkčnosť.

7.4.1 `AbstractProcess`

Trieda `AbstractProcess` dedí od triedy `Event`. Vďaka tomu je možné ju plánovať. Ďalej definuje rozhranie dôležité pre proces: abstraktné metódy `suspend()` a `terminate()` slúžiace na pozastavenie a ukončenie procesu. Pre používateľa definuje metódy `waitFor()` a `passivate()`, ktoré sú vlastne iba kombináciou plánovania a pozastavenia procesu. Metóda `waitFor()` pozastaví proces na dobu simulačného času určenú parametrom. Metóda `passivate()` pozastaví proces, kým nebude znova naplánovaný. Tieto metódy môžu byť zavolané iba nad práve vykonávaným procesom, v opačnom prípade bude vyvolaná výnimka `SimException`.

7.4.2 `ThreadProcess`

Trieda `ThreadProcess` dedí od triedy `AbstractProcess`. Implementuje simulačný proces pomocou vlákna. Zmenu stavu modelu popisuje metóda `behavior()`, rovnako ako v triede `Event`. Na rozdiel od triedy `Event`, tu metóda `behavior()` beží vo vlastnom vlákne, vďaka čomu je možné ju pozastaviť a obnoviť. Trieda `ThreadProcess` má odkaz na objekt predstavujúci toto vlákno (`Thread thisThread`). Pomocou tohto objektu môže vlákno riadiť. Trieda

`ThreadProcess` má ďalej dva objekty triedy `Object` (`mainThreadLock` a `thisThreadLock`). Monitory týchto objektov sa využívajú pri kooperatívnom prepínaní vláken. Vždy, keď sa pozastavuje vlákno tohto procesu a prebúdza sa vlákno riadenia simulácie, najprv vlákno tohto procesu vojde do monitora objektu `thisThreadLock`. Potom vojde aj do monitora objektu `mainThreadLock`, aby mohlo prebudiť vlákno riadenia simulácie a prebudí ho. Opustí monitor objektu `mainThreadLock` a ostane čakať na monitore objektu `thisThreadLock`, ktorý ešte neopustilo. Tento monitor vlákno opustí až po tom, ako čakanie dokončí. Ukážka kódu:

```
synchronized(thisThreadLock) {
     synchronized(mainThreadLock) {
          mainThreadLock.notifyAll();
     }
     thisThreadLock.wait();
}
```

Prepínanie vláken je takto implementované z dôvodov popísaných v kapitole 6.2.1. Prepínanie vláken v opačnom smere prebieha analogicky, iba úlohy vláken sú opačné.

7.4.3 `OperationProcess`

Trieda `OperationProcess` dedí od triedy `AbstractProcess`. Implementuje proces bez použitia vlákna. Zmenu stavu modelu popisuje zoznam operácií. Trieda `Operation` pre objekty operácií je vnorená v triede `OperationProcess`. Proces sa nedá prerušiť uprostred operácie, iba medzi nimi (operácie sú atomické). Trieda `OperationProcess` má pole objektov triedy `Operation` (`program`) a informáciu o tom, ktorá operácia sa práve vykonáva udržuje vo forme indexu do tohto poľa (`operationPointer`). Používateľ môže zoznam operácií určiť pomocou metódy `setProgram()`. Táto metóda je implementovaná tak, že zoznam operácií nastaví iba pri prvom zavolaní, a to z dôvodu ochrany, pretože definícia procesu by sa počas simulácie nemala meniť. Po spustení procesu sa operácie vykonávajú zaradom (`operationPointer` sa inkrementuje). Poradie operácií je možné zmeniť pomocou metódy `jump()`. Táto metóda nastaví `operationPointer` podľa svojho parametra. Takže po jej vykonaní sa dokončí aktuálna operácia a pokračuje sa operáciou na novom indexe. Keď sa má proces pozastaviť, dokončí sa aktuálna operácia a proces sa pozastaví až za ňou. Proces skončí, keď `operationPointer` dôjde na koniec poľa operácií.

7.5 Ostatné triedy simulačného jadra

Implementácia ďalších tried sa veľmi nelíši od štandardných riešení (implementácie v SIMLIB/C++), preto popíšem iba prípadné rozdiely.

- `Facility`

Implementácia bez prerušenia obsluhy.

- `Store`
- `Stat`
- `TStat`

Pridaná metóda `record(double x, boolean incN)`, pomocou ktorej sa dá špecifikovať, či sa má inkrementovať počítadlo záznamov, alebo nie.

- Generátory náhodných čísel

Metódy `uniform()` a `exponential()` sú v triede `Simulation`.

8 Testovanie

V tejto kapitole popisujem testy vyvinutého simulačného nástroja a porovnanie ich výsledkov s testami simulačného nástroja SIMLIB/C++. Testoval som najmä pamäťovú a časovú efektivitu.

8.1 Príklad použitia

V balíčku `example` je príklad použitia vyvinutého simulačného nástroja. Je v ňom implementovaný jednoduchý model samoobsluhy. Model je implementovaný pomocou oboch prístupov: pomocou vláken, aj bez vláken. To, ktorý prístup sa použije pri simulácii, treba určiť pri spúšťaní programu argumentom na príkazovom riadku. V adresári `simlib-c++` je ten istý model implementovaný v SIMLIB/C++. Z výsledkov simulácií vidno, že nástroje sú ekvivalentné.

8.2 Popis testov

V balíčku `test` je implementovaný jednoduchý test výkonu vyvinutého simulačného nástroja. Model je jednoduchý systém hromadnej obsluhy M/M/5. Pri štarte programu je možné pomocou argumentov na príkazovom riadku nastaviť strednú dobu medzi príchodmi požiadaviek, strednú dobu obsluhy, časové obmedzenie simulácie a ďalšie parametre. Pri štarte programu sa musí pomocou argumentov špecifikovať, či sa majú použiť simulačné procesy triedy `ThreadProcess` alebo `OperationProcess`, a či sa má spustiť test optimalizovaný pre meranie využitia pamäte alebo pre meranie spotreby času.

Počas testu pre meranie využitia pamäte beží simulácia raz a v pravidelných intervaloch (dĺžku intervalu je možné špecifikovať argumentom pri spustení programu) sa do výstupného súboru zapíše záznam s položkami:

- `simTime` simulačný čas
- `realTime` reálny čas v nanosekundách (od spustenia simulácie)
- `processCount` počet procesov v systéme
- `virtualMemory` množstvo virtuálnej pamäte, ktorú využíva systémový proces Java Virtual Machine
- `maxMemory` maximálna kapacita heap v Java Virtual Machine
- `totalMemory` aktuálna kapacita heap v Java Virtual Machine
- `freeMemory` voľná pamäť na heap v Java Virtual Machine
- `usedMemory` využitá pamäť na heap v Java Virtual Machine

Čas a využitie heap sa meria pomocou štandardných nástrojov, ktoré Java ponúka (`System`, `Runtime`), ale údaj `virtualMemory` sa číta z pseudosúboru `/proc/self/stat`, takže tento údaj bude dostupný iba v operačných systémoch založených na UNIX-e. Vždy pred zápisom záznamu sa spúšťa garbage collector, aby bolo možné zistiť, koľko pamäte na heap zaberajú užitočné dáta. Spúšťanie garbage collector a čítanie pseudosúboru `/proc/self/stat` simuláciu výrazne spomaľuje. To je hlavný dôvod, prečo sú testy na meranie pamäte a času rozdelené.

Počas testu pre meranie spotreby času beží simulácia s rovnakými parametrami niekoľkokrát (počet opakovaní simulácie je možné špecifikovať argumentom pri spustení programu) a do výstupného súboru sa zapíše iba jeden záznam pre každý beh simulácie. Pri tomto teste sa nemeria pamäť, preto záznam obsahuje iba prvé tri položky z vyššie uvedeného zoznamu, pričom položka `realTime` udáva dobu trvania simulácie v reálnom čase.

V adresári `simlib-c++` je implementovaný ekvivalentný test pre SIMLIB/C++. Pri spustení programu je možné pomocou argumentov na príkazovom riadku špecifikovať rovnaké parametre. Do výstupného súboru sa zaznamenávajú tie isté položky, okrem informácií o využití heap. Položka `virtualMemory` vypovedá o množstve virtuálnej pamäte, ktoré využíva systémový proces, v ktorom beží simulácia. Čas sa zisťuje pomocou funkcie `clock_gettime()` z `time.h` a `virtualMemory` sa číta z pseudosúboru `/proc/self/stat`.

8.3 Výsledky testov

Výsledky testov sú závislé od platformy, nad ktorou simulačný nástroj beží a na zložitosti simulácie. Preto slúžia iba na porovnanie jednotlivých nástrojov a prístupov. Ja som testy vykonával na stroji s procesorom Intel Pentium 1.73 GHz a s 1 GB pamäte.

8.3.1 Maximálny počet procesov v systéme

Model som nastavil tak, aby množstvo simulačných procesov v systéme neustále rástlo a sledoval som, pri akom množstve simulácia havaruje.

- Prístup pomocou vláken

Simulácia bežala bez problémov, kým v systéme nebolo približne 8000 procesov. Potom bola vyvolaná výnimka, ktorá informovala o tom, že nie je možné vytvoriť ďalšie natívne vlákno.

- Prístup bez vláken

Simulácia bežala bez problémov, kým v systéme nebolo približne 145000 procesov. Potom sa vyčerpala všetká pamäť na heap a simulácia havarovala. Maximálna kapacita heap sa dá

nastaviť pri štarte Java Virtual Machine. Pri zväčšení tejto kapacity sa úmerne zväčšil aj maximálny počet procesov v systéme.

- SIMLIB/C++

Simulácia bežala, kým sa nevyčerpala systémová pamäť. Počet procesov v systéme prekročil milión.

8.3.2 Závislosť spotreby pamäte od počtu procesov v systéme

Model som nastavil tak, aby množstvo simulačných procesov v systéme neustále rástlo a spustil som test pre meranie využitia pamäte. Sledoval som závislosť spotreby pamäte od počtu procesov.

- Prístup pomocou vláken

Pri štarte simulácie proces Java Virtual Machine využíval približne 206 MB virtuálnej pamäte. Počas behu simulácie množstvo virtuálnej pamäte využitej procesom Java Virtual Machine rástlo s každým novým simulačným procesom v systéme približne o 335 KB.

Pri štarte simulácie bolo využitých približne 147 KB pamäte na heap. Počas behu simulácie množstvo pamäte využitej na heap rástlo s každým novým simulačným procesom v systéme približne o 385 B.

- Prístup bez vláken

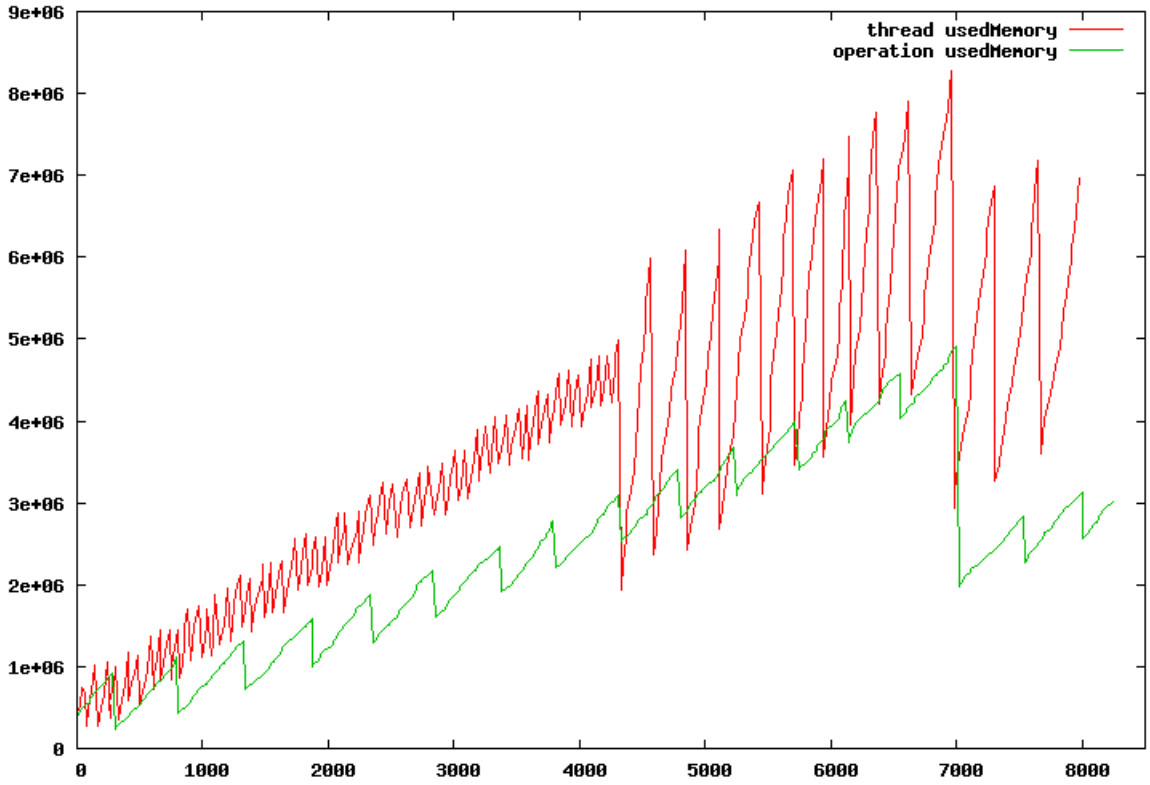
Pri štarte simulácie proces Java Virtual Machine využíval približne 206 MB virtuálnej pamäte. Táto hodnota ostala konštantná počas celej simulácie.

Pri štarte simulácie bolo využitých približne 147 KB pamäte na heap. Počas behu simulácie množstvo pamäte využitej na heap rástlo s každým novým simulačným procesom v systéme približne o 245 B.

- SIMLIB/C++

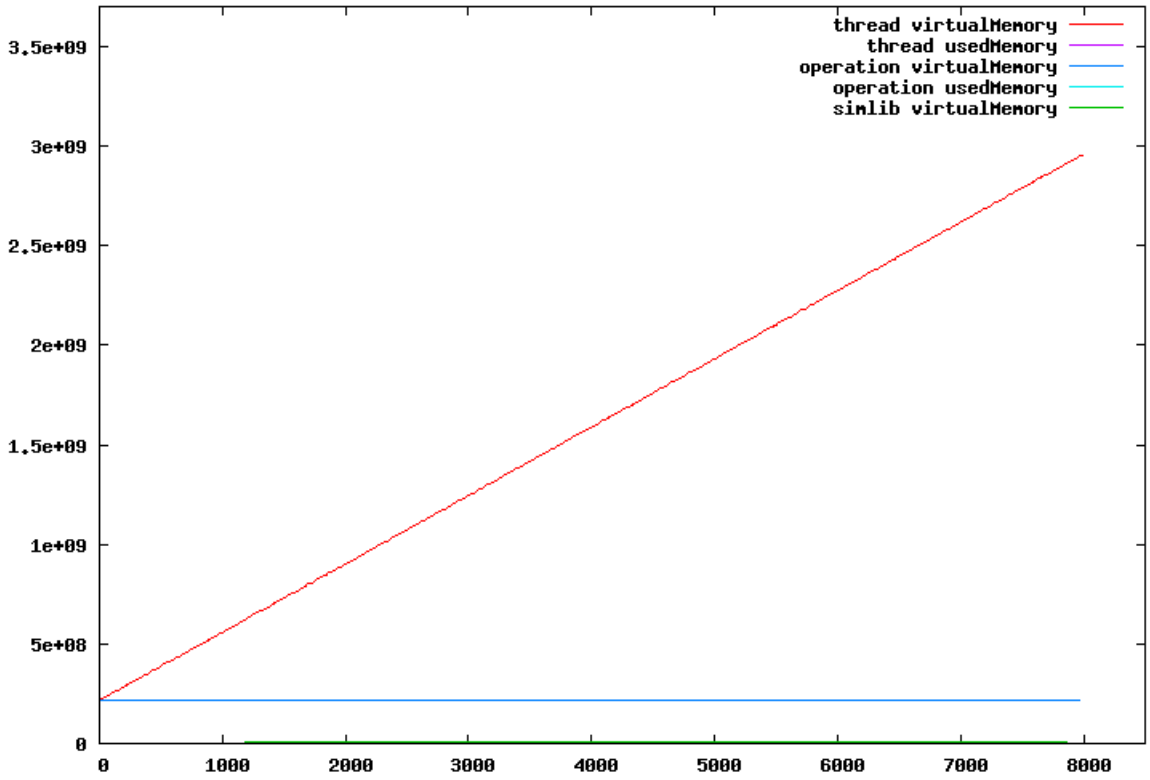
Pri štarte proces simulácie využíval približne 3368 KB virtuálnej pamäte. Počas behu simulácie množstvo virtuálnej pamäte využitej procesom simulácie rástlo s každým novým simulačným procesom v systéme približne o 498 B.

Z hodnôt nameraných pri týchto testoch som vyhotovil nižšie uvedené grafy. Na grafoch názorne vidno výsledky testovania.



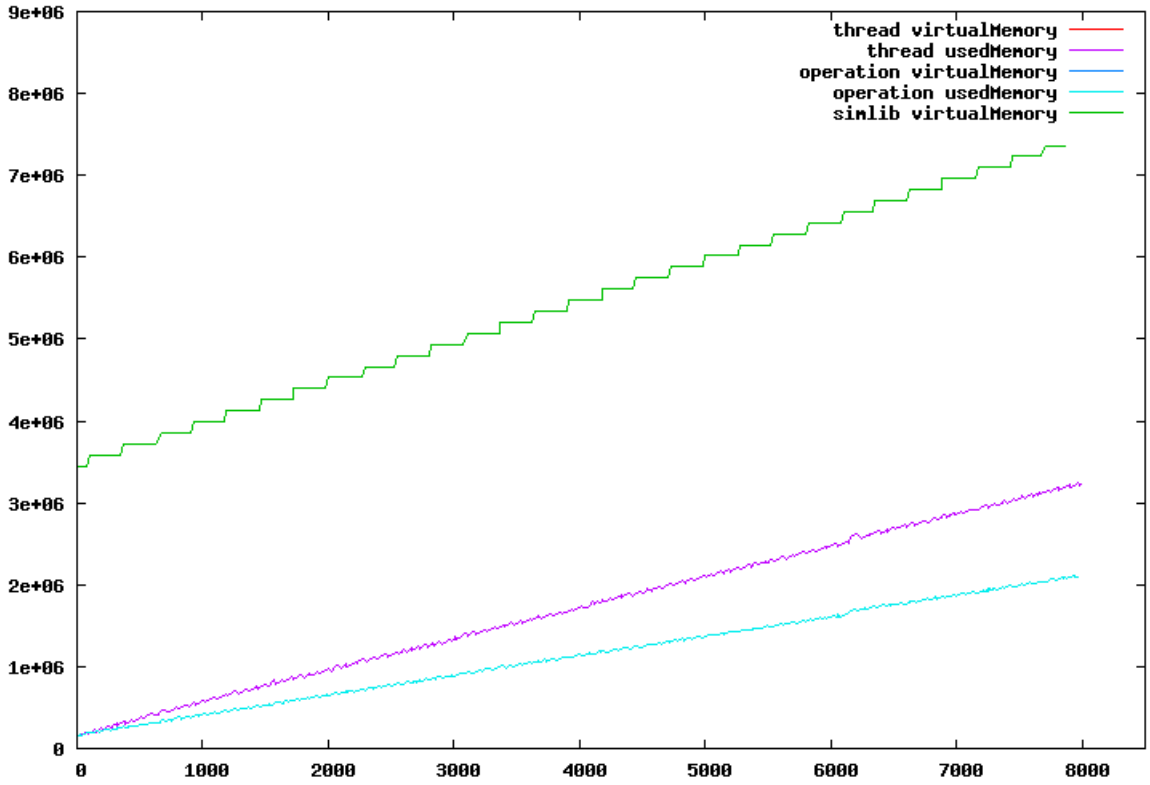
Graf 8.1: Závislosti využitia pamäte na heap od počtu procesov v systéme meraná bez použitia garbage collector

Tento graf zobrazuje závislosti využitia pamäte na heap od počtu simulačných procesov v systéme. Vodorovná os udáva počet simulačných procesov a zvislá množstvo využitej pamäte v bytoch. Červená čiara popisuje hodnoty získané počas simulácie s využitím simulačných procesov implementovaných pomocou vláken a zelená popisuje hodnoty získané počas simulácie s využitím simulačných procesov implementovaných bez vláken. Hodnoty zobrazené v tomto grafe boli merané bez volania garbage collector, preto vykazujú určitú nepravidelnosť.



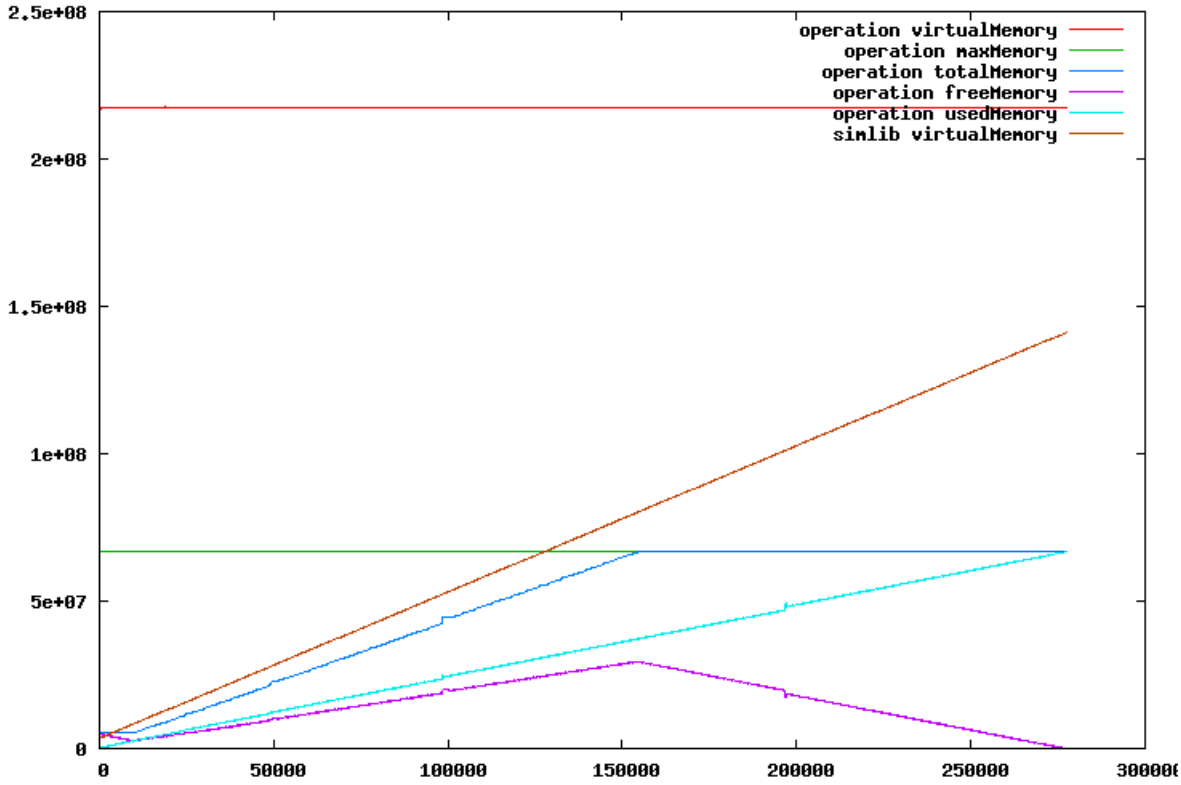
Graf 8.2: Závislosti využitia pamäte od počtu procesov v systéme

Tento graf zobrazuje závislosti využitia pamäte od počtu simulačných procesov v systéme. Vodorovná os udáva počet simulačných procesov a zvislá množstvo využitej pamäte v bytoch. Červená čiara zobrazuje závislosť množstva virtuálnej pamäte, ktorú využíval proces Java Virtual Machine pri behu simulácie s využitím simulačných procesov implementovaných pomocou vláken, a tmavomodrá zobrazuje závislosť množstva virtuálnej pamäte, ktorú využíval proces Java Virtual Machine pri behu simulácie s využitím simulačných procesov implementovaných bez vláken. Ostatné hodnoty zobrazené v tomto grafe sú príliš nízke na to, aby ich bolo vidno, a budú viditeľne zobrazené v nasledujúcom grafe.



Graf 8.3: Závislosť využitia pamäte od počtu procesov v systéme (zväčšená)

Tento graf zobrazuje tie isté závislosti, ako predchádzajúci. Líši sa od neho iba v tom, že je zväčšený tak, aby bolo vidieť najnižšie hodnoty. Zelená čiara zobrazuje závislosť množstva virtuálnej pamäte, ktorú využíval proces simulácie pomocou knižnice SIMLIB/C++. Fialová čiara zobrazuje závislosť využitia heap počas simulácie s využitím simulačných procesov implementovaných pomocou vláken a bledomodrá zobrazuje závislosť využitia heap počas simulácie s využitím simulačných procesov implementovaných pomocou vláken.



Graf 8.4: Závislosť využitia pamäte od počtu procesov v systéme (dlhá simulácia)

Tento graf zobrazuje závislosti využitia pamäte od počtu simulačných procesov v systéme. Vodorovná os udáva počet simulačných procesov a zvislá množstvo využitej pamäte v bytoch. Graf bol zhotovený z hodnôt získaných pri simulácií, ktorá bežala dostatočne dlho na to, aby sa minula pamäť na heap (čo na grafe vidno). Červená čiara zobrazuje závislosť množstva virtuálnej pamäte, ktorú využíval proces Java Virtual Machine pri behu simulácie s využitím simulačných procesov implementovaných bez vláken. Zelená zobrazuje závislosť maximálnej kapacity heap, tmavomodrá závislosť aktuálnej kapacity heap, fialová závislosť množstva voľnej pamäte na heap a bledomodrá závislosť množstva využitej pamäte na heap počas simulácie s využitím simulačných procesov implementovaných bez vláken. Hnedá čiara zobrazuje závislosť množstva virtuálnej pamäte, ktorú využíval proces simulácie pomocou knižnice SIMLIB/C++.

8.3.3 Doba trvania simulácie

Model som nastavil tak, aby simulácia netrvala príliš dlho, ale aby cez systém prešlo dostatočné množstvo procesov. Desaťkrát som spustil test pre meranie spotreby času so sto opakovaniami simulácie. Zo všetkých nameraných hodnôt som potom vypočítal aritmetický priemer.

- Prístup pomocou vláken

Jeden beh simulácie trval približne 1,261 sekúnd.

- Prístup bez vláken

Jeden beh simulácie trval približne 0,034 sekundy.

- SIMLIB/C++

Jeden beh simulácie trval približne 0,017 sekundy.

8.4 Vyhodnotenie testov

Z vykonaných testov jednoznačne vyplýva, že prístup pomocou vláken je podstatne pomalší a pamäťovo náročnejší ako prístup bez vláken. SIMLIB/C++ je v porovnaní s vyvinutým simulačným nástrojom v Jave rýchlejší a má výhodu v tom, že nepotrebuje Virtual Machine, ktorá zaberá veľa systémovej pamäte. Pri testovaní proces Java Virtual Machine vždy zaberal približne o 200 MB viac pamäte ako bolo alokované pre heap. Na druhú stranu simulačné procesy vyvinutého simulačného nástroja zaberali na heap približne o polovicu menej pamäte ako simulačné procesy v SIMLIB/C++. Je možné, že by po aplikovaní optimalizácií mohol vyvinutý simulačný nástroj vyrovnať pamäťový handicap, ktorý získal kvôli Virtual Machine.

9 Záver

Výsledkom tejto práce je diskrétny simulačný nástroj vyvinutý pomocou programovacieho nástroja Java, ktorý porovnáva dva prístupy k implementácii procesov: s použitím vláken a bez použitia vláken.

Z testovania vyplynulo, že prístup pomocou vláken je viditeľne pomalší a pamäťovo náročnejší ako prístup bez vláken. Na druhú stranu prístup pomocou vláken je používateľsky príjemnejší. Algoritmus popisujúci dej v modelovanom systéme je jednoducho napísaný v určitej metóde v programovacom jazyku Java. Podobné používateľské rozhranie by sa dalo pri použití prístupu bez vláken dosiahnuť pomocou externého skriptu, alebo prepisom (inštruktážou) skompilovaných súborov, v ktorých je implementovaný model.

V porovnaní so simulačnou knižnicou SIMLIB/C++ bol vyvinutý nástroj viditeľne pomalší. Porovnať pamäťové nároky je o čosi zložitejšie, pretože Java Virtual Machine zaberá veľa systémovej pamäte. Počas behu simulácie na vyvinutom nástroji využíval systémový proces Java Virtual Machine konštantné množstvo virtuálnej pamäte, ktoré bolo pomerne vysoké. Oproti tomu množstvo virtuálnej pamäte využitej systémovým procesom, v ktorom bežala simulácia v SIMLIB/C++, priamo úmerne záviselo od množstva simulačných procesov v simulácii. Množstvo pamäte, ktoré zaberali simulačné procesy vyvinutého nástroja na heap v Java Virtual Machine bolo asi o polovicu nižšie, ako množstvo virtuálnej pamäte, ktoré zaberali simulačné procesy knižnice SIMLIB/C++. Po aplikovaní optimalizácií by mohol vyvinutý simulačný nástroj vyrovnať pamäťový handicap, ktorý získal kvôli Virtual Machine.

V budúcnosti by stálo za úvahu otestovať účinnosť niekoľkých optimalizácií vyvinutého simulačného nástroja, napríklad vlastná implementácia front procesov a kalendára tak, že dáta potrebné pre vloženie elementu do fronty/kalendára by boli súčasťou elementu. Rovnako by sa oplatilo otestovať simulačný nástroj JiST a preskúmať možnosti, ktoré ponúka inštruktáž skompilovaných tried (pozri kapitolu 3.6).

Literatúra

[1] Peringer, Petr: *Modelování a simulace, Studijní opora*, Brno, FIT VUT v Brně, 28.11.2006
[2] *SIMLIB Home Page* [online], Posledná modifikácia: 16. 12. 2008 [cit. 2009-05-20], Dostupné na URL: <http://www.fit.vutbr.cz/~peringer/SIMLIB/>
[3] Peringer, Petr: *Porting SIMLIB/C++ to 64-bit Platform*, In: Proceedings of XXIXth International Autumn Colloquium ASIS 2007, Ostrava, CZ, MARQ, 2007, s. 155-160, ISBN 978-80-86840-34-5
[4] *Simkit Home Page* [online], Posledná modifikácia: 11. 08. 2008 [cit. 2009-05-20], Dostupné na URL: <http://diana.nps.edu/Simkit/>
[5] *DESMO-J* [online], Posledná modifikácia: 21. 02. 06 [cit. 2009-05-20], Dostupné na URL: <http://desmoj.sourceforge.net/home.html>
[6] *SimJava* [online], [cit. 2009-05-20], URL: <http://www.dcs.ed.ac.uk/home/hase/simjava/>
[7] *J-Sim Home Page* [online], Posledná modifikácia: 28. 01. 2005 [cit. 2009-05-20], Dostupné na URL: <http://www.j-sim.org/>
[8] *JiST - Java in Simulation Time* [online], [cit. 2009-05-20], URL: <http://jist.ece.cornell.edu/>
[9] Venners, Bill: *Inside the Java Virtual Machine*. McGraw-Hill, 2000, ISBN 0-07-135093-4
[10] *JDK 6 Documentation* [online], [cit. 2009-05-20], URL: <http://java.sun.com/javase/6/docs/>

Zoznam príloh

Príloha 1. CD so zdrojovými textami, programovou dokumentáciou a nameranými hodnotami