УДК 519.683.4

К.т.н., доцент Марченко О.І., студент Зінгерман Е.В.

Національний технічний університет України
«Київський політехнічний інститут»

ДЕТЕРМІНОВАНИЙ ЗА ЧАСОМ МЕТОД АВТОМАТИЧНОГО КЕРУВАННЯ ПАМ'ЯТТЮ НА БАЗІ МЕХАНІЗМУ ПІДРАХУНКУ ПОСИЛАНЬ

Abstract

Oleksandr Marchenko, assoc prof., PhD; Eduard Zingerman, student
Time determined method of automatic memory management based on reference counting technique

This paper presents the time determined algorithm of automatic memory management based on reference counting technique which is capable to handle cyclic data structures while not using garbage collector. The main difference of the algorithm from traditional reference counting is usage of number of possible paths from lexical context to the object, instead of reference count.

Вступ

Засоби автоматичного керування пам'яттю широко використовуються при розробці прикладного програмного забезпечення, що значно спрощує процес розробки та унеможливлює цілий клас помилок. Водночас вони не поширені в системному програмному забезпеченні та в системах реального часу (СРЧ). Це зумовлено непередбачуваними затримками при виконанні процесу пошуку блоків пам'яті, що були виділені однак більше не мають посилань з контексту виконання програми (“збору сміття”), та обмеженнями, які накладають на структури даних алгоритми керування пам'яттю, що не потребують збирання сміття.

В даній статті пропонується алгоритм підрахунку посилань, що може коректно працювати з циклічними посиланнями та має детермінований час виконання.

Постановка задачі

Задача полягає в розробці алгоритму підрахунку посилань, який

коректно працює з циклічними структурами даних та має детермінований час виконання.

Огляд існуючих розв'язків

Механізм підрахунку посилань є досить розповсюдженим в програмних системах, наприклад у ядра ОС Linux, де цей механізм широко використовується для структур даних, що описують пристрої, буфери вводу/виводу, елементи файлової системи, тощо.

Крім того, підрахунок посилань широко розповсюджений в системах збирання сміття, наприклад у мові програмування Python. В цих системах підрахунок посилань дозволяє оперативно звільнювати "ациклічне" сміття, в той час як решту збирає повноцінний алгоритм збирання сміття.

Таку ж ідеологію сповідує і решта розповсюджених алгоритмів керування пам'яттю на базі підрахунку посилань [1][2][3].

Слід зазначити, що фаза збирання сміття може вносити суттєву, а головне недетерміновану, затримку в процес виконання програми.

Більшість сучасних досліджень спрямована на мінімізацію кількості оновлень лічильників посилань та на розпаралелювання алгоритму.

Опис алгоритму

Обмеження алгоритму підрахунку посилань при роботі з циклічними структурами даних може бути проілюстроване наступним прикладом (рис. 1).

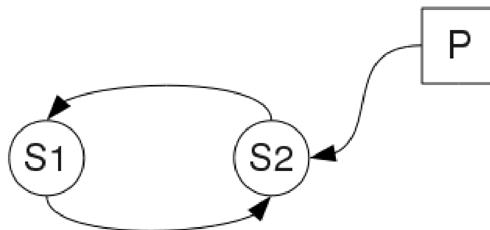

Рис. 1

S1 та S2 — об'єкти, що містять посилання один на одного, P — вказівник з лексичного контексту програми на об'єкт S2. Лічильник посилань об'єкту S1 дорівнює 1, лічильник посилань об'єкту S2 дорівнює 2. Коли процес виконання залишить контекст програми, в якому оголошений вказівник P, лічильник об'єкту S2 зменшиться до одиниці. Таким чином

пам'ять обох об'єктів залишиться не звільненою, в той час як вони стали недосяжними для програми, тобто станеться витік пам'яті (memory leak).

Для звичайного підрахунку посилань вірно твердження — нульове значення лічильника посилань однозначно вказує на те, що об'єкт є недосяжним з жодного лексичного контексту програми, ненульове значення лічильника посилань гарантує досяжність об'єкту з лексичних контекстів тільки за умови відсутності циклічних посилань.

Автори пропонують підраховувати не самі посилання, а *кількість шляхів з існуючих лексичних контекстів програми* до об'єкту. Тоді нульове значення такого лічильника однозначно вказує на те, що об'єкт є недосяжним з жодного лексичного контексту, а ненульове значення гарантує досяжність об'єкту як мінімум з одного лексичного контексту програми.

Алгоритм підрахунку кількості шляхів розглянемо на базі наступної моделі.

Довільний об'єкт S містить в собі лічильник Cnt, прапорець V та множину вказівників [P].

Далі у тексті використовуються наступні позначення та операції:

1. P — довільний вказівника;
2. S.<текст> - звернення до поля об'єкта з ім'ям <текст>;
3. P = new S — створення нового об'єкту;
4. P1 = P2 — копіювання вказівника;
5. P1 = S1.P2 — копіювання вказівника з об'єкту;
6. S1.P1 = P2 — копіювання вказівника з контексту в об'єкт.

Програма складається з наступної послідовності дій:

1) оголошення вказівників;
2) послідовність операцій;
3) виконання операції deref (дивись нижче) для кожного оголошеного вказівника.

Кожна з наведених операцій замінюється наступним псевдокодом:

Операція	**Псевдокод**

`P = new S`	`deref(P);`
	`P := addr(S);`
	`S.Cnt = 1;`

`P1 = S1.P2`
`P1 = P2`
	`deref(P1);`
	`P1 := P2;`
	`propogate-cnt-inc(P1, 1);`

S1.P1 = P2	deref(P1);
	P1 := P2;
	propogate-cnt-inc(P1, S1.Cnt);

deref(P)	if not P->V
	      P->V = true
	      foreach p from P->[P]:
	            deref(p);
	      P->V = false;
	
	P->Cnt -= 1;
	if (P->Cnt == 0)
	      free(P);

propogate-cnt-inc(P, val)	if not P->V
	      P->V = true
	      P->Cnt += val;
	      foreach p from P->[P]:
	            propogate-cnt-inc(p, val);
	
	      P->V = false

Твердження. Всі об'єкти, які були створені під час виконання програми, коректно звільнені після її завершення. Це твердження може бути доведене методом математичної індукції, однак доведення є предметом окремої статті.

За допомогою запропонованої моделі можна описати всі операції над вказівниками в імперативних мовах програмування типу C та Pascal, за винятком арифметики вказівників.

Оцінка алгоритму та можливі оптимізації

Кожна з операцій має складність порядку O(n), де n — кількість елементів на які прямо чи опосередковано посилається об'єкт.

Водночас, в більшості випадків можливі суттєві оптимізації. Як приклад розглянемо процес додавання елементу до однозв'язного списку (рис. 2а, 2б).

а) до виконання операції б) після виконання операції

Рис. 2. Додавання елементу в список

Програма яка виконує цю операцію має наступний вигляд:

Вихідний текст	Після перетворення

P1 = S1
P2 = P1.next
P3 = S3
 	
P1.next = P3
P3.next = P2
	# P1 = S1
	deref(P1);
	P1 := S1;
	propogate-cnt-inc(P1, 1);
	# P2 = P1.next
	deref(P2);
	P2 := P1.next;
	propogate-cnt-inc(P2, 1);
	# P3 = S3
	deref(P3);
	P3 := S3;
	propogate-cnt-inc(P3, 1);
	
	# P1.next = P3
	deref(P1.next);
	P1.next := P3;
	propogate-cnt-inc(P1.next, P1.Cnt);
	# P3.next = P2
	deref(P3.next);
	P3.next := P2;
	propogate-cnt-inc(P3.next, P3.Cnt);
	deref(P1) ;
	deref(P2) ;
	deref(P3);

В такій реалізації можна нарахувати 6 проходів по списку. Водночас, якщо відокремити операцію звільнення пам'яті від операції deref, то операції propogate-cnt-inc та deref стають комутативними.

Очевидно, що в такому разі, пари “propogate-cnt-inc(X, 1) — deref(X)” можна виключити з коду, зводячи кількість проходів по списку до трьох, крім того виконання додаткового аналізу посилань може зменшити кількість проходів до виконання однієї операції propogate-cnt-inc, яка може не зробити жодного проходу по списку якщо її аргумент є нульовим.

Ключем для можливих оптимізацій є комутативність операцій propogate-cnt-inc та deref, за умови, що з останньої винесено фазу звільнення пам'яті.

Порівняння з існуючими рішеннями

Існуючі рішення проблеми підрахунку циклічних посилань, такі як, наприклад, підрахунок циклічних посилань з відкладеним маркуванням та

скануванням (“Cyclic Reference Counting with Lazy Mark-Scan”) включають в себе фазу пошуку та збирання сміття, що зумовлює недетермінізм часу виконання, неприйнятний в системах реального часу.

Висновки

Запропонований алгоритм може бути прийнятним рішенням для роботи з структурами даних, що не передбачають занадто великої кількості розгалужень, таких як дескриптори сокетів, списки буферів вводу/виводу тощо.

Напрямком подальших досліджень є створення компілятору діалекту мови програмування С, в який включено додаткові мовні конструкції для використання запропонованого методу керування пам'ятю, з метою оцінювання ефективності цього алгоритму оптимізації для конкретних застосувань.

Література

1. *Rafael D. Lins* “Cyclic Reference Counting with Lazy Mark-Scan” - Computing Lab. - The University - Canterbury — England 1990
2. *Rafael D. Lins* “Generational Cyclic Reference Counting” - Computing Lab. - The University - Canterbury — England 1992
3. *Rafael Dueire Lins, Francisco Heron de Carvalho Junior, Zanoni Dueire Lins* “Cyclic Reference Counting with Permanent Objects” — Journal of Universal Computer Science, vol. 13, no. 6 (2007), 830-838
4. *Andrei Alexandrescu* “Modern C++ Design: Generic Programming and Design Patterns Applied” — Addison Wesley, February 01, 2001
5. *David F. Bacon, V.T. Rajan* “Concurrent Cycle Collection in Reference Counted Systems” — Proc. European Conf. on Object-Oriented Programming, June, 2001, LNCS vol. 2072