Списание „Математика, компютърни науки и образование“ Том 7, кн. 2, 2024
Journal of Mathematics, Computer Science and Education Vol. 7, No. 2, 2024

DOI: 10.54664/FPKY9556

АТОМЕН[1] МОДЕЛ НА ПАМЕТТА

Мирослав Бончев

ATOMIC MEMORY MODEL

Miroslav Bonchev

Abstract: Memory leaks and buffer overruns are among the most common erroneous conditions in computer systems. They are also among the most tedious, difficult and time consuming to detect, identify, locate, and resolve. Both Microsoft and Google independently report that memory safety issues cause 70% of all security vulnerabilities [10]. The Atomic Memory Model provides a mechanism for resolving these types of errors. Simultaneously, it makes the software architecture more robust and elegant by enforcing a code hierarchy regarding the memory use, as well as allowing for innate memory abstraction that matches the application semantics. The model also significantly increases the speed of the software development by relieving developers from detailed memory management.
Keywords: Memory atoms, memory leaks, buffer overruns, memory safety, speed of development, quality of code

ВЪВЕДЕНИЕ

Паметта в компютърните системи представлява група от клетки за данни с еднакъв размер, всяка от които е уникално идентифицирана чрез линеен адрес. В модерните системи, на физическо ниво организацията на паметта може да има значително по-сложен характер за идентификация и

[1] Относно термина „Atomic Memory Model“: В техническата литература е прието „atomic“ да се превежда на български език като „атомарен“ и като „атомен“, само когато става въпрос за неща, свързани с „ядрен“. Това е в несъответствие с оригиналната дума „átomos”, която означава първо „индивид, човек“, след това „неделим“ и на последно място е заимствана от дефинициите на Leucippus и Democritus за „атом“ в смисъла на неделима частица, изграждаща света. Моделът на паметта, предложен в тази статия, дефинира затворени, неделими частици памет със специфични характеристики, които нарича „атоми памет“, като по този начин ясно и точно описва тяхната същност – индивидуални и неделими. „Атом памет“, както е дефиниран в тази работа, е в пълно съответствие, както с оригиналната дефиниция на думата като „индивид“ и „неделим“, така и със заимстваното от Leucippus и Democritus значение като частица изграждаща съответния свят на приложението. От друга страна, ако моделът се нарича „атомарен“, то за да има съответствие, формациите, които моделът предлага, би трябвало да се наричат „атомари памет“ а не „атоми памет“. Авторът предоставя на читателя да прецени и реши как да нарича модела на български език: „атомарен модел на паметта“, както би било правилно по конвенция, или „атомен модел на паметта“, както е в съответствие със семантиката на думата „átomos“ (атом). На английски език, на който моделът бе първоначално създаден, името е Atomic Memory Model, дефиниращ memory atoms.

достъп, включително пейджинг, сегментиране, кеширане, двоен достъп и др. На приложно ниво обаче паметта винаги се предоставя на приложенията като едномерен масив от линейни адреси [11, 4], сочещи към уникално идентифицирани клетки памет [9]. Обикновено те имат размер един байт. И така всеки бит в цялото пространство на паметта е уникално идентифициран и достъпен чрез адреса на клетката памет и неговата позиция в нея, както е показано на Фиг. 1.

Първи адрес: 0	…	X	X+1	...	X+n	максимален адрес

Фиг. 1. *Изглед на памет както се предоставя на приложенията*

Операционната система по принцип е собственикът на паметта. Приложенията искат памет от операционната система и са отговорни да ѝ я върнат, когато тя повече не им е необходима. Приложението е отговорно също за това да използва дадената му памет коректно, включително и да не излиза от нейните граници. При неспазване на тези условия се получават грешки в паметта, които обикновено водят до катастрофални резултати [10, 8]. В някои езици като C/C++ програмистът е отговорен за освобождаването на паметта, използвана от приложението, докато в други като C# за това е отговорна системата на езика за „събиране на боклуци“ (garbage collection) [2].

Изложение

Приложенията заемат памет от операционната система (ОС) чрез извиквания на съответните нейни функции. Паметта се предоставя като едномерен масив с начален адрес и с размер от 0 до n байта, където поискан размер памет $\leq n \leq$ адресируема или налична памет, в зависимост от типа на системата. Технически вземането на памет от ОС е само временно прехвърляне на собствеността на предадения масив, който е част от глобалния масив памет на процеса, от ОС към приложението. Прехвърлянето е просто предаване на адреса на първия елемент на масива на приложния код, като ОС си води вътрешен отчет за това. По конвенция приложния код приема, че „получената“ памет е масив от последователни, непрекъснати, уникално разпознаваеми клетки памет с размер поне толкова, колкото са поискани. Предадената памет не се премества, изолира, защитава, прехвърля или нещо друго. Системата за управление на паметта само маркира предадените адреси като притежавани от приложението и не ги използва, докато не ѝ бъдат върнати. Предадената памет няма тип – тя е просто масив от последователни адреси с необходимия размер. Например, ако повикващият поиска памет с размер 3 байта, той може да получи адрес X, като в езиците за програмиране C и C++ това може да изглежда по следния начин:

```
// Задели ми 3 байта памет.
void* p_memory = malloc(3);
```

При успех на операцията, p_memory има стойност X. Системата за управление на паметта маркира вътрешно байтове с адреси X, X+1 и X+2 като притежавани от приложния код и не ги използва, докато не ѝ бъдат върнати чрез съответното извикване. Понякога предаденият блок може да бъде по-голям от поискания.

```
// Връщам блока от 3 байта памет.
free( p_memory );
```

Приложението е отговорно за:
- Връщане на паметта, когато вече не е необходима.
- Гарантира, че не чете или пише извън блока памет, който му е даден.

Когато приложението не изпълни тези изисквания, то създава едно или повече от следните състояния на грешка:

• Изтичане на памет – приложението не е върнало непотребната за него вече памет на операционната система. Причините за това могат да бъдат:
- o непълен или неправилен алгоритъм;
- o грешка в изпълнението на алгоритъма;
- o изтичане на памет по време на обработка на изключение;
- o паметта се освобождава на грешна система за разпределение на паметта;
- o паметта се освобождава в грешен куп.

• Фрагментация на паметта – това е следствие на изтичането на памет.

• Превишаване на буфера – приложението чете или пише извън блока памет, който му е даден. Причините за това могат да бъдат:
- o непълен или неправилен алгоритъм;
- o грешка в изпълнението на алгоритъма.

• Срив на приложението – в някои системи и/или в някои случаи горните състояния на грешка могат да доведат до изключение за нарушение на достъпа, или друг тип състояние на срив на приложението или системно изключение, особено когато:
- o паметта се освобождава на грешна система за разпределение на паметта;
- o паметта се освобождава в грешен куп;
- o паметта се чете/записва от/на място извън разрешената за четене/запис страница/сегмент;
- o паметта се чете/записва от/на място вътре в разрешената за четене/запис страница/сегмент, но на неправилен адрес, като по този начин причинява състояние на срив (незабавно или по-късно), например деление на нула.

Грешки с паметта от този вид са едни от най-трудните, неприятни и отнемащи време за решаване, тъй като те изискват:

1. Осъзнаване – много от проблемите с паметта могат да останат скрити за дълго време и може да се проявяват само в редки състояния на приложението.

2. Идентифициране – да се определи, че проблемът е именно повреда в паметта. Често може да се окаже трудно да се определи точната причина за грешката. Например много труден за идентифициране проблем може да бъде, когато при неправилни изчисления един байт в масив от данни е повреден.

3. Локализиране – определянето на адреса или адресите на повредата, както и увреждащия код също може да бъде изключително труден проблем. Особено в многонишкови приложения или ако проблемът се появява само в Release компилации.

4. Разрешаване – веднъж локализиран, проблемът може да бъде лесно решен, например корекция на неправилно отместване. Но има случаи, когато решаването на проблема изисква съществени усилия и пренаписване на значителни количества код.

Други трудности и проблеми, които са по-рядко срещани включват:

1. Трудности за контрол на достъпа до блокове памет в многонишкови приложения – тази трудност се проявява когато повече от една нишка се опитват да пишат и четат от един и същ буфер памет [5]. В такива случаи писането и четенето от паметта трябва да се синхронизира. Тъй като всяка от нишките използва само един указател, който сочи към буфера памет, то синхронизацията трябва да се организира с външни за паметта средства.

2. Липса на контекст на блоковете памет при отстраняване на грешки – тъй като паметта се идентифицира само с указателя към нея, то към нея няма прикачена метаданни, като например кой, къде, кога, защо я е заделил и по късно освободил. Така при отстраняване на грешки блоковете памет не могат да се идентифицират лесно с програмното изпълнение.

Систематизация и анализ

Систематизирайки по-горе изложената информация виждаме, че:

1. Паметта се заделя с извикване на функция.
2. Паметта се освобождава с извикване на функция, като е от особена важност:

a. използването на правилната функция, и

b. на правилното място.

3. Указателят към паметта се използва директно от приложния код.
4. Указателят към паметта е единственото нейно свойство, известно на приложния код.
5. Паметта няма физически граници.
6. Заделената паметта няма стриктен тип, освен когато е заделена с оператор “new”, но и тогава типът ѝ може да се променя или да бъде само частично стриктен.
7. Типът на паметта може да се манипулира/реинтерпретира по всяко време.

Обобщавайки, можем да изразим главните проблеми с паметта:

1. Паметта принадлежи на системата [4, 11], а системата разчита на приложението да я управлява коректно, т.е. най-висока важност системен софтуер разчита на най-ниска важност приложния софтуер да управлява системните ресурси коректно.
2. Паметта е дефинирана като едномерна линейна структура, която се интерпретира и реинтерпретира като многомерна абстрактна структура, т.е. тя се манипулира като аморфна маса, което лесно може да доведе до грешки.

И така, проблемите с паметта се дължат основно на използването ѝ по примитивен начин и без абстракция.

Атоми памет

Предлаганият модел на паметта се състои в създаването на подходяща абстракция на паметта, която след необходимото тестване до нивото на системен софтуер може да се използва от приложенията с цел предотвратяване на проблемите с паметта.

Дефиниция: <u>Атом памет</u> (<u>Memory Atom</u>) е затворена формация, която представлява абстрактна памет с *клас* и *семантика*, съдържаща (сурова) памет с *произход*, и предоставяща услуги с памет на приложението чрез интерфейс. Свойствата на атома памет (memory atom) са:

1. Клас – дава специализация и гранулираност на атома. Класът на атома памет указва типа на съдържащите се в него елементи. Например специализирането на един атом памет като BYTE означава, че той съдържа байтове. Атомите памет могат да бъдат специализирани с всеки, включително потребителски дефиниран тип.
2. Семантика – дава специфично (включително полиморфно) поведение на атома памет. Семантиката на атома памет определя природата му, как ще се държи, какви характеристики ще притежава, какви специализирани интерфейси ще излага. Например атом памет, дефиниран като многонишков, определя семантиката на безопасен достъп до съдържащата се е него памет от няколко нишки едновременно. „Сигурен“ атом памет определя семантика, че съдържанието на паметта се изтрива при освобождаване на паметта и т.н.
3. Произход – дава системата памет, която в крайна сметка е собственик на паметта предоставена на атома памет. Произхода указва откъде ще бъде заделена паметта и съответно къде по-късно освободена. Например паметта може да бъде заделена от процесната памет, от друга системна купчина, от COM, от C библиотека, а може изобщо да не бъде заделена/освободена, например „Shell“ памет и т.н.

Дефиниция: <u>Типът</u> на един атом памет се определя от неговите: клас, семантика и произход.

Езиците за програмиране от високо ниво, способни да дефинират потребителски типове, могат да използват атоми памет за подобряване на качеството на кода, написан на тях. Обектно ориентирани езици с детерминистично управление, и способности за наследственост и специализация, като C++, са обаче най-подходящи за приложението на Атомарния/Атомния/Atomic модел на памет, и могат да използват всички ползи и предимства, които той предоставя.

При използването на обектно ориентиран език за програмиране, когато приложението създаде атом памет, то неговият конструктор заделя необходимото количество памет от съответната система за памет. Когато атом памет бъде разрушен, то неговият деструктор освобождава паметта, която той притежава с помощта на подходящата функция за освобождаване на памет отново в съответната система за памет. Така програмистът е освободен от задължението да освобождава

памет, която вече не му е необходима, като за това вече се грижи компилаторът, включително и при обработката на изключения.

Тъй като паметта, която атомът притежава, е достъпна само чрез методи, то с използване на подходящ защитен код в методите на атомите, те гарантират, че няма да допуснат препълване на буфера или друга грешка в паметта. В случай, че те открият грешка обаче, атомите сигнализират кода, който ги използва по стандартен начин за възстановяване на системата или грациозно спиране, като генерират изключение или връщат код за грешка за по-нататъшна обработка. Това гарантира изключително стабилна система по отношение на обработката на паметта, като в същото време кодът е компактен и подреден.

В допълнение, за да се избягнат числови и контекстни грешки, атомите памет се броят, индексират, числено пресмятат и отнасят изключително и само в „единици памет“.

Единици памет

Дефиниция: Единица памет (Memory Unit) е цяло число с определен *клас*, който я специализира и указва класа на атомите памет, към които тя може да се отнася.

Единиците памет се използват за преброяване, добавяне, изваждане, сравняване на размера, промяна на размера, индексиране и други числови операции с и на атомите памет и техните елементите. Атомите памет не разпознават числа, освен единици памет от същия клас като тях. Например атом памет от клас DWORD може да бъде принуден да предостави стойността на 25-ият DWORD елемент в него, само с помощта на единица памет от клас DWORD със стойност 24 (индексирането започва от 0). Използването на общ целочислен тип за математически операции с атоми памет е забранено, тъй като:

1. Безтипови числа биха деградирали абстракцията на модела.

2. Когато се използват атоми памет по хетерогенен начин, например при сериализация, единиците памет дават средства за определяне на размера на съдържащата се от атомите памет в байтове или в други класове. Това е изключително полезно, като прави кода ясен, интуитивен и компактен. В противен случай, при използване на общи цели числа, ще трябва всички изчисления и конвертирания да се правят изрично, което премахва абстракцията и дава широки възможности за грешки.

3. В случай, че общи числа могат да се използват за аритметични операции с атомите памет и техните елементите, ще има обърквания кога едно цяло число се използва като единица памет, кога като байт, кога като нещо друго. Например ако се използва цяло число за Allocate(const uint units) вместо специализиран тип Allocate(const Unit< class > units), то може лесно да се получи грешка и програмистът да пресмята в друг клас памет, например в байтове, вместо в необходимия клас. Често препълвания на буфери и други подобни грешки са причинени от такива недоразумения.

Закон: Единици памет от класа на атома памет са единственият начин програмата да се отнася аритметично към него и неговите елементи.

Изключения

При откриване на грешки атомите и единиците памет генерират изключения или сигнализират за получилата се грешка по друг подходящ начин. Тъй като подготовката на стека за функции, които генерират изключения, е ресурсоемко, то за приложения, които използват паметта много интензивно е неподходящо да генерират изключения в атомите и единиците памет, тъй като това може да забави цялостното изпълнение. В такива случаи използването на ASSERT (УСЛОВИЕ), вместо if (УСЛОВИЕ) then throw (exception) обикновено е достатъчно за решаването на проблема, тъй като моделът стимулира ранно откриване на грешките, и ASSERT-ите ги прихващат по време на разработката на кода. В програми с нормално интензивно използване на паметта това не е необходимо, тъй като в такива случаи забавянето в настройките на стека е неусетно за потребителя.

Право на съществуване

В Атомарния/Атомния/Atomic модел на паметта, от гледна точка на приложението, паметта съществува единствено и само като атоми памет от определен тип, т.е. капсулирани самоподдържащи се единици с клас, семантика и произход, и никога като „памет“. В рамките на атома, „в неговото ядро“, където абстракцията е „съблечена“, атомът използва паметта по традиционния начин. За външния свят обаче атомът памет е капсулиран обект, който е достъпен само чрез предоставен от него интерфейс.

Пространство на съществуване

Атомите памет освобождават заетата от тях памет в деструкторите си. По тази причина, за да се изпълни автоматично деструкторът на един атом памет, при което той да освободи заеманата от него памет, е необходимо те да бъдат обхванати от рамка на стека. Затова атомите памет или се създават в стека директно, или по някакъв начин се съдържат от обект, който в крайна сметка е създаден в стека, и който е отговорен за тяхното унищожаване. Такъв обект може да бъде например интелигентен показалец или свързан списък, който е създаден на стека, и съдържа атоми памет създадени в Heap паметта на процеса. По такъв начин пространството на съществуването на атомите памет трябва да бъде винаги директно или индиректно стекът. Когато рамката на стека, където атомите памет съществуват, бъде освободена, независимо дали нормално или поради развиването му след изключение, то пространството на съществуването им се унищожава и всички съдържащи се там обекти също се освобождават автоматично или директно или чрез деструктора на обекта, които ги съдържа, както в примера с интелигентен показалец и свързания списък. От изключително важно значение е атомите памет винаги да са пряко или непряко свързани с рамка на стека, така че при излизането от нея паметта да бъде автоматично освободена.

Атоми памет дефинирани в по-ниски рамки на стека, могат да се предават на по-горни рамки като формални параметри на функциите, който дефинират тези по-горни рамки. В тези функции и методи те могат да бъдат използвани според необходимостта. В този случай, е ясно, че атомите памет могат да бъдат разрушени/освободени само при излизане от рамката на стека където са дефинирани. Въпреки, че в по-горните рамки те не могат да бъдат освободени, там те могат да бъдат манипулирани, включително да бъде променена или освободена паметта, която съдържат.

Атоми памет дефинирани в обекти на стека, които ги съдържат, също ще бъдат освободени при излизане от рамката на стека, където е дефиниран обектът, който ги съдържа. В този случай обаче, обекта, който ги съдържа може да ги създава и разрушава динамично във всяка рамка на стека, по-горна от тази където той е дефиниран. Такова използване дава една важна гъвкавост за използването на атомите памет, при това без да се губи нищо от ползите и сигурността, която атомите памет дават на кода, при условие, че съдържащия ги обект е добре дефиниран.

Правила на атомарния/атомния/atomic модел на паметта

1. Атомите памет са абстракция на капсулирана памет с клас, семантика и произход.
2. Атомите памет позволяват само защитен достъп до паметта, която притежават.
3. Атомите памет използват защитаващ код в своите методи [7, 8, 10], покриващ възможните състояния на грешки за метода.
4. Атомите памет генерират изключения или сигнализират за грешки [7, 8] по друг начин.
5. Единици памет от класа на атома памет са единственият начин за аритметическо отнасяне към него.
6. Атомите памет съществуват само в рамка на стека директно, или индиректно чрез съдържащ ги обект на стека, който е отговорен за тяхното съществуване.
7. Приложенията работят само с атоми памет и никога директно с паметта.

Допълнителна функционалност и йерархия

Разрешаването на грешките с паметта е само една от ползите от въвеждането и използването на предложения модел на паметта. При използването на пълния спектър от инструменти и възможности на обектно ориентираното програмиране могат да се получат съществени ползи от гледна точка на качеството на кода, включително подобряване на архитектурата на приложението, чистотата, простотата и компактността на кода, както и скоростта за създаването му, и скоростта на работата му. Използвайки наследяване може да се дефинира набор от полезни общи задължителни операции, които всички типове атоми на паметта поддържат. Общ абстрактен родител налага задължителен интерфейс, чрез който се налагат способности към всички типове атоми памет, така дефинирайки семейство/а от типове. Това дава огромни предимства включително:

• При необходимост, идентификация на всеки атом памет, така че грешки могат да се проследят много по-лесно отколкото без идентификация на паметта.

• Записване и четене на атоми памет.

• Значително опростяване разработването и поддръжката на атомите памет с използването на наследственост.

• Лесна подмяна и надграждане на използваните атоми памет при необходимост по време на компилация.

• Позволява създаването на гъвкави интерфейси и API използващи атоми памет за параметри или резултати.

• Използването на специализация на атомите памет допълнително позволява тяхното комбиниране за получаването на композитно поведение според необходимостта на приложението.

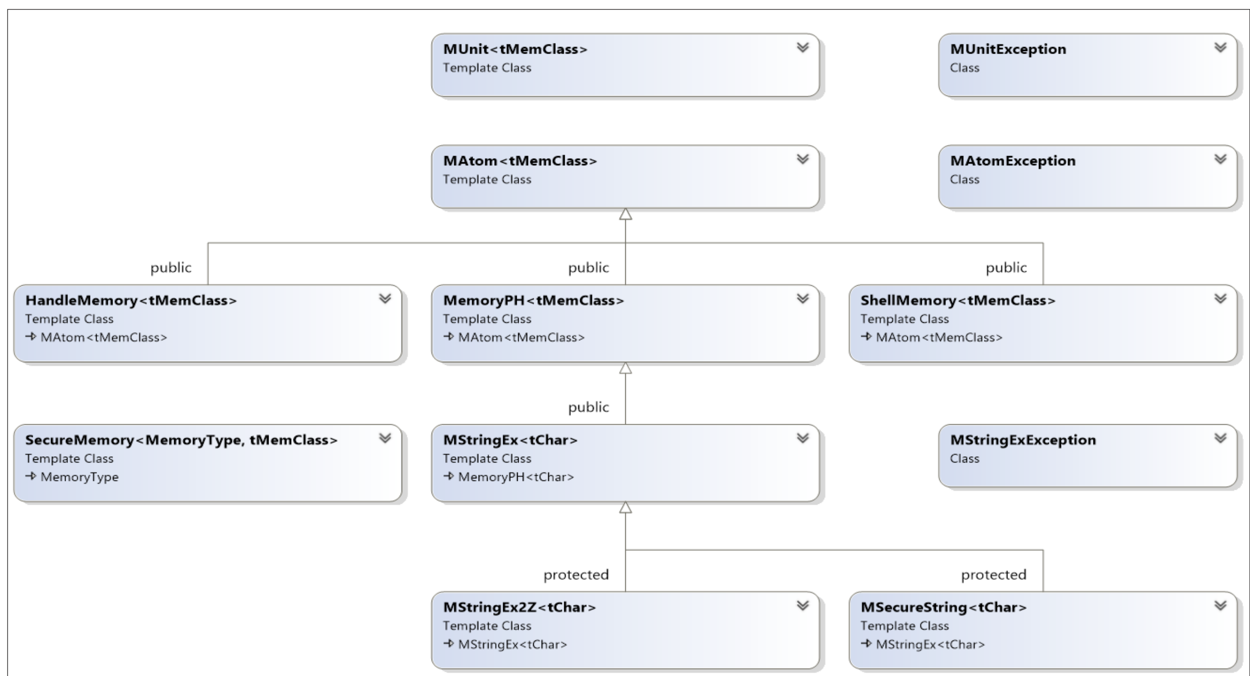
Дефинирането на методи и оператори за събиране, нулиране, сравняване, обръщане, записване, зареждане и други позволява допълнително бързо и качествено разработване на софтуер, използвайки атомите памет. Спецификацията на атомите памет по начин на заемане и връщане на паметта, както и по начина на работа допълнително улесняват разработчиците и ускоряват тяхната дейност, например: атоми памет със стандартна и поточна работа, съединена и сегментирана вътрешна структура, едномерна и многомерна памет, защитена и незащитена семантика, еднонишкова и многонишкова семантика, и т.н.

Специфично в C++, от версия 11 (2011 г.) насетне се препоръчва използването на контейнери [11, 6], като vector< class >, list< class >, vector< byte > и други, вместо масиви от обекти или „гола“ памет. Допълнително, още от края на 90-те години на XX век, Microsoft използват широко smart pointers (умни указатели) и RAII принципа в MFC и ATL [1, 6]. Като пример за такъв умен указател наближаващ вече 30 годишна възраст ще посочим CComPtr< class >, а за тип използващ RAII принципа lock за CRITICALSECTION. Microsoft са дефинирали стотици такива типове в тези и други свой библиотеки. По-късно стандартната библиотека на C++ приема тези идеи и също започва да препоръчва използването на тези техники. Тук ще отбележим, че умните указатели и RAII са едно и също нещо по-идея и същество, като разликата между тях е само една степен на индиректност. Действително подхода с използването на контейнери решава проблема с изтичането на паметта и нарушаването на границите ѝ, а използването на RAII предпазва от изтичането и на други ресурси [1, 6]. Обаче, както практиката показва, решаването на проблема с изтичането и границите на паметта, за които AMM бе създаден първоначално преди контейнерите на stl, е една много малка част от предимствата, които атомите памет дават. Със своята абстракция, те правят програмирането, не само свободно от грешки на изтичане и корупция в паметта, но и съществено по-качествено, по-бързо, по-лесно, по-гъвкаво, по-силно и по-чисто от други типове грешки. Например дефинираме атоми, които действително управляват паметта, както и такива, които предоставят само изглед към съществуваща памет. Тогава, клиента може да поиска изглед на паметта от съдържащ я атом, който да бъде плъзган по нея, така отнасяйки се към различни части от „истинската“ памет. Прозорецът също може да произведе под-прозорец от друг вид (клас) памет и така потоци памет могат да бъдат лесно създавани и интерпретирани. Като добавим гъвкавостта давана от свойството „произход“ (например главен Heap, вторичен Heap, COM

и т.н.), както и тази давана от свойството „семантика“, например secure/non-secure, thread-unsafe/thread-safe и т.н. може да се видят лесно огромните предимства на модела над контейнерите на std и RAII, в които няма абстракция [1, 6].

Примерно изпълнение и използване

Atomic Memory Model бе първоначално създаден през 2001 година [3] с цел решаване на практически трудности, свързани с грешки в паметта в практическа среда. След интензивна разработка и ползване, през 2005 година бе създадена втора итерация и имплементация на модела, като се взеха предвид откритите недостатъци в първоначалните идеи и дизайн. Примерно изпълнение на Atomic Memory Model версия 2.4 може да бъде изтеглена и използвана от https://github.com/MiroslavBonchevBonchev/Atomic_Memory_Model. Опростена йерархия от атоми памет в този код е показана на Фиг. 2.



Фиг. 2. *UML диаграма на атоми памет дефинирани в имплементация v.2.4*

Тук MUnit< tMemClass > е единица за памет. MAtom< tMemClass > е абстрактен родител за всички атоми памет. MemoryPH< tMemClass > е най-често използваният атом памет, който заделя и използва памет от общата памет на процеса. ShellMemory< tMemClass > е атом памет, който не заделя и не освобождава памет, а работи като прозорец в други видове памет, включително в собствения си вид. ShellMemory< tMemClass > е много удобен за работа с потоци, тъй като показва и работи с определена част от потока. MStringEx< tChar > е стринг, който наследява от MemoryPH< tMemClass >, така стрингът е просто нула терминиран специализиран блок обикновена памет. MStringEx2Z< tChar > е двойно нула-терминиран стринг. SecureMemory< MemoryType, tMemClass > и MSecureString< tChar > нулират съдържанието на паметта преди да я върнат на собственика ѝ, и т.н.

За използването на модела и предоставения код, потребителят само трябва да включи съответните .h файлове, от които се нуждае в собствения си код, например:

```
#include “AMM/MAtom.h”          // или #include “AMM/StringEx.h”
```

В кода атомите памет се декларират като обикновени обекти и се използват според необходимостта. Например, в кода на Фиг. 3 дефинираме атоми памет на стека и записваме съдържанието им във файлове:

```
MemoryPH< BYTE >    mem( MUnit< BYTE >( 10 ), 0xFF );

mem.SaveAsFile( "filename_10.bin", 0, nullptr );
```

```
filename_10.bin
00000000  FF FF FF FF FF FF FF FF  FF FF                    ..........
```

```
ShellMemory< BYTE > m22 = mem.SubMemory( MUnit< BYTE >( 2 ), MUnit< BYTE >( 2 ) );

m22.XOR( 0xAA );

m22.SaveAsFile( "filename_22.bin", 0, nullptr );
```

```
filename_22.bin
00000000  55 55                                             UU
```

```
mem = mem + m22 + mem;

mem.SaveAsFile( "filename_final.bin", 0, nullptr );
```

```
filename_final.bin
00000000  FF FF 55 55 FF FF FF FF  FF FF 55 55 FF FF 55 55  ..UU......UU..UU
00000010  FF FF FF FF FF FF                                 ......
```

Фиг. 3. *Използване на атоми памет в C++ програма*

В примера показан на Фиг. 4 създаваме няколко атома памет на стека, като предпазваме приложението от грешка в паметта с try-catch блок. Различните части от имплементацията генерират собствени изключения за да направят по-лесна диференциацията на грешките с паметта, ако се получат такива.

В същия примера функцията do_some_work_on_the_data(params), приема като аргумент атом памет, и връща като резултат атом памет. Приемащата функция може да промени или не подадения като параметър атом памет. Тя може да промени съдържанието на паметта, да е освободи, да и промени размера и т.н. Върнатия атом, може да съдържа „истинска“ памет или да бъде прозорец в „истинска“ памет. Модела поддържа както copy-конструктори, така и move-семантика, което го прави допълнително компактен и силен.

Функцията fun_print(params) разпечатва 32-битови положителни числа. В случая ma2 е поток, за който да кажем знаем, че има поне 4 DWORDS след първия байт. Методът SubMemory< specifier >(params) създава временен атом памет на стека от тип ShellMemory< DWORD >, който не заема и не освобождава памет а само я показва, съдържащ 4 DWORDS след първия байт на ma2. Това е изключително бърза операция тъй като памет не се копира. След оптимизациите компилаторът ще махне всичко излишно и ще остави само един указател и един размер [3]. Тогава fun_print(params) ще използва временния ShellMemory< DWORD > атом памет за да разпечата съдържанието му. В случай, че ma2 не може да произведе 4 DWORDS след първия байт, например по някаква причина ma2 има по малък размер, то той ще генерира изключение или ASSERT.

```
try
{
   // Create a DWORD array on the stack.
   DWORD dwa[3] = { 1, 2, 3 };

   // Create a memory atom coping the content of the stack array.
   MemoryPH< DWORD > ma4( dwa, MUnit< DWORD >( 3 ) );

   // Call the function passing ma4 to it, and assign the returned memory to ma2.
   MemoryPH< BYTE > ma2 = fun_do_some_work_on_the_data( ma4 );

   // Print the middle 4 DWORDS of m2, starting from the second byte of the memory.
   fun_print( ma2.SubMemory< DWORD >( MUnit< BYTE >( 1 ), MUnit< DWORD >( 4 ) ) );
}
catch( const MUnitException e )
{
   printf( TEXT( "Memory unit exception." ) );
}
catch( MAtomException e )
{
   printf( TEXT( "Memory atom exception." ) );
}
catch( MStringExException e )
{
   printf( TEXT( "String exception." ) );
}
catch( ... )
{
   printf( TEXT( "Unknown exception." ) );
}
```

Фиг. 4. *Използване на атоми памет в C++ програма в защитен блок*

Тъй като съществуват голям брой API, който не са запознати с идеята за Atomic Memory Model, примерната имплементацията по-горе намалява строгостта на някои от правилата му, за да може той да бъде използван по-широко в практиката.

Въпреки, че примерите в статията са дадени на езика C++, методологията не е подчинена на този специфичен език, и може да бъде успешно приложена във всяко обкръжение до съответната степен, в което има автоматична инициализация и де-инициализация на обектите при тяхното създаване и разрушаване, наследственост и специализация на обектите.

ЗАКЛЮЧЕНИЕ

Атомарният/Атомният/Atomic модел на паметта успешно решава проблемите свързани с използването на памет в програмни системи без автоматично управление на паметта, като C++ програми и други. Включително:

1. Няма изтичане на памет – при развиването на стека паметта се освобождава автоматично от деструкторите на обектите.
2. Няма препълвания на буфери и нарушаване на техните граници – гарантира се от методите на атомите памет.
3. Дава атомичен (атомен) достъп до паметта в многонишкови приложения – гарантира се от семафори в методите на атомите памет.
4. Идентификация на паметта при дебъгване:

- Лесно може да се постави уникален идентификатор на всеки атом памет.
- Лесно може да се добави метаинформация и символи към атомите памет, например файла и реда, кой заема, променя, и освобождава всеки атом и неговата памет, и т.н.
- Лесно може да се създадат лог файлове за трасиране на грешките, както във времето при всяко действие върху атом памет, така и като дъмп на метаинформация и символи запазени в тях.

В допълнение Атомарният / Атомният / Atomic модел на паметта дава серия от други ползи и предимства пред стандартно използване на паметта и стандартните методи за освобождаването ѝ [3]:

- простота на програмирането;

- чистота на кода;
- оптималност на изпълнението;
- абстрактност и гъвкавост;
- стимулира създаването на по-стабилна и елегантна софтуерната архитектура;
- стимулира абстракция на паметта, съответстваща на семантиката на приложението;
- съществено увеличаване на качеството на разработката;
- съществено увеличаване на скоростта на разработката;
- в редки случаи на проблеми с паметта, дава възможност за много по лесно дебъгване;
- детерминираност на освобождаването на паметта;
- прави концепцията за събиране на отпадъци памет (Garbadge Collection) безсмислена.

ЛИТЕРАТУРА

[1.] **Alexandrescu, A. (2001).** Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Professional.

[2.] **Boehm, H. J. and Weiser, M. (1988).** “Garbage Collection in an Uncooperative Environment.” Software: Practice and Experience 18, 9, 807–820.

[3.] **Bonchev, M. (2009).** “Atomic Memory Model.” Information Technologies and Control, 2/2009, ISSN 1312-2622.

[4.] **Drepper, U. (2007).** “What Every Programmer Should Know About Memory.” Red Hat, Inc. Technical Report.

[5.] **Herlihy, M. and Shavit, N. (2008).** The Art of Multiprocessor Programming. Morgan Kaufmann.

[6.] **Meyers, S. (2014).** Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O’Reilly Media.

[7.] **Miller, M. and Triplett, J. (2019).** “Mutation Testing for Memory Safety.” ACM SIGPLAN Notices 54, 4, 199–213.

[8.] **Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. (2009).** “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C.” ACM SIGPLAN Notices 44, 6, 245–258.

[9.] **Pike, R., et al. (1995).** “Plan 9 from Bell Labs.” Computing Systems 8, 3, 221–254.

[10.] **Seacord, R. (2013).** Secure Coding in C and C++, 2nd Edition. Addison-Wesley Professional.

[11.] **Stroustrup, B. (2013).** The C++ Programming Language, 4th Edition. Addison-Wesley Professional.

ИНФОРМАЦИЯ ЗА АВТОРА

Мирослав Бончев, асистент, докторант,
специалност „Информатика“, Факултет „Математика и информатика“,
Великотърновски университет „Св. св. Кирил и Методий“,
e-mail: m.bonchev@ts.uni-vt.bg

ABOUT THE AUTHOR

Miroslav Bonchev, assistant, PhD student in Informatics,
Faculty of Mathematics and Informatics, ”St. Cyril and St. Methodius”
University of Veliko Tarnovo, e-mail: m.bonchev@ts.uni-vt.bg