Лекция 16 (17. Януари, 2012)

Links
http://en.wikipedia.org/wiki/Tail_call
http://en.wikipedia.org/wiki/Inline_function
http://en.wikipedia.org/wiki/Static_variable
http://en.wikipedia.org/wiki/Volatile_variable
http://java.sun.com/developer/technicalArticles/Networking/HotSpot/inlining.html

Първи час

В днешно време компилаторите вършат значителна част от работата на програмистите. Преди време, обаче, това не е било така. Преди, например програмистите не са можели да използват ефективно всички възможности на процесорите, освен ако не са познавали в детайли архитектурата, за която пишат. Дори прости неща като смяна на реда на изпълнение на два независими реда код би могло в някои случаи да доведе до големи печалби откъм време за изпълнение. В днешно време за наше щастие това не е нужно, тъй като компилаторите са ужасно умни (без ваше знание те променят кода ви значително преди да го превърнат в машинен език, което води до страхотни подобрения в бързодействието на кода). Все пак бихме могли да постигнем известни печалби ако познаваме по-добре архитектурата на процесора (но определено не толкова, колкото преди). И въпреки всичко и днес има случаи, в които можем да подобрим бързодействието на програмите, които пишем, с бегли познания от компютърни архитектури. Прост пример за това е следната задача: дадени са ви две матрици от числа (да кажем квадратни) – умножете ги. Разбира се, имплементацията на този алгоритъм не е особено трудна – всеки от вас трябва да знае как да напише три цикъла, които умножават матриците по познатия ни начин „ред по стълб". В часовете по Алгебра са поменати и други сходни начини – „ред по ред" или „стълб по стълб", но тъй като те де факто правят същото ги използваме много рядко (нали сме си свикнали с „ред по стълб"). И докато за нас това е абсолютно еквивалентно, и на пръв поглед за компютъра също трябва да е абсолютно еквивалентно (в крайна сметка броят извършени операции е абсолютно същия), на практика се оказва, че ако не умножаваме матриците „ред по стълб" ами „ред по ред" понякога постигаме (при достатъчно големи матрици) двойно и повече подобрение в скоростта. Откъде идва тази шокираща разлика? От начина, по който компютърът оперира с паметта! При четене на една клетка от матрицата, компютърът всъщност чете и няколко клетки след нея. При следваща заявка за някоя от тези клетки той я връща веднага (тя е някъде из кешовете), вместо да рови отново в паметта (което не е твърде бързо). Това води до бързо четене на ред от матрицата (тъй като клетките са последователни) и бавно четене на колона от матрицата (тъй като клетките вече не са последователни). Сега, връщайки се на двата на пръв поглед еквивалентни алгоритъма за умножение на матрици, виждаме, че в „ред по стълб" имаме „бързо по бавно", а в „ред по ред" имаме „бързо по бързо". От тук идва и над двойното подобрение в скоростта – от проста оптимизация на операциите с паметта. Друг (подобен) пример за оптимизация на скоростта е ако индексираме масив и просто сменим реда на измеренията му.

Например вместо int a[MAX_N][MAX_M] да го направим int a[MAX_M][MAX_N]. Това, в зависимост от задачата, би могло отново да промени скоростта на изпълнение осезаемо (с над 10%). Когато пишем performance-critical част от кода, понякога тези 10% могат да са от голямо значение). В Java е възможно да правим дори по-абсурдни оптимизации. Тъй като там няма „многомерни" масиви като при C/C++, а вместо това има масиви от pointer-и към други масиви, то при индексиране в 4-мерен масив реално имаме 4 проследявания на pointer вместо 1, както бихме имали в C/C++. Това доста често бави работата на програмата, ако тя е много memory-intensive. Можем да покажем как да „симулираме" C/C++ масиви в Java. Нека сме имали четиримерен масив a[N][M][S][T]. Той има N * M * S * T на брой клетки. Вместо него, можем да заделим едномерен масив с размер b[N * M * S * T]. За да индексираме клетка a[i][c][j][k] в масива b[], то трябва да вземем неговата клетка b[i * (M * S * T) + c * (S * T) + j * T + k]. На пръв поглед правим много повече изчисления, но реално достъпът в паметта е многократно по-бавен от умноженията и събиранията, така че по този начин можем да спечелим скорост (в една от задачите, където ми се наложи да правя това, постигнах подобрение около два пъти). Разбира се, много зависи колко често е индексирането, колко размерен е масивът, колко голям е масивът и други.

Тези операции, обаче, биха били контра-интуитивни за човек, незапознат с това как работи паметта. По подобен начин има детайли на процесора (а и като цяло на цялото нещо, наречено „компютър"), чрез които бихме могли да изкрънкаме допълнителна скорост, тъй като компилаторът не винаги е достатъчно умен, да ги оптимизира (други пример за това е буферираното четене и писане, което разгледахме преди няколко лекции).

Друга хитра оптимизация, която компилаторът прави невидимо за вас, е така наречената „опашечна рекурсия" (tail recursion). Това е фунцкия, която вика себе си (recursion) като последна логическа операция (tail). Показано с код, това е:

```
int factorial(int n)
{
	if (n <= 1)
		return 1;
	return n * factorial(n – 1);
}
```

Функцията може да има някакъв код в тялото си, както и да вика други функции, но оптимизацията се прилага при последното викане – тъй като нямаме изпълняване на други операции след извикването на функцията, вместо да се заделя памет за адрес за връщане (тоест къде да се върне кодът след като приключи с извиканата функция), тя се връща директно в бащата на factorial(n) (тоест функцията, която е извикала нея). Нещо повече, в случая на рекурсия, тъй като аргументите вече не се използват, те могат да бъдат преизползвани в следващото викане (тоест във factorial(n – 1)), вместо да бъдат заделяни наново. Това, ако се замислите, превръща рекурсията в цикъл! Също така спестява известна памет в стека, като понякога помага избягването на stack overflow. Така компилаторът превръща вашата рекурсия в код, който не е рекурсивен! Красота! ☺ Има известни изисквания за да може компилаторът да осъществи тази оптимизация, като най-важното от тях е викането да е в return-а (или поне да е последна логическа операция в рекурсията). Ако се интересувате повече от нея (а и от нейната

генерализация tail call, която важи не само за рекурсивни функции), можете да погледнете статията в Wikipedia.
В следствие на горните примери, а и на по-професионалното програмиране от това в университета, в езиците има някои думи, които почти не се споменават и ползват по време на изучаването на програмните езици. В тази лекция ще разгледаме не много добре познатите кодови думи „inline", „static", и „volatile". Ролята на всяка от тях е специфична и това е една от причините да са малко познати сред начинаещите програмисти.

Static
Тази кодова дума има различни значения в C/C++ и Java. Ще разгледаме и двете.
Статична променлива е памет, която е заделена по време на цялото изпълнение на програмата (тоест се заделя при нейното стартиране и се освобождава при exit или return от main). Тя се заделя в нормалната памет на компютъра и потенциално може да е много голяма като размер. Най-лесният начин да заделите статична променлива е просто да я декларирате извън тялото на функция. За разлика от тях, останалите променливи се заделят в стека по време на изпълнение на програмата. Съществуват няколко други разлики между двата типа: примерно статичните променливи са гарантирано инициализирани с нула (тоест паметта, върху която са заделени, е нулирана преди заделянето им), за разлика от заделените в стека, където това не се гарантира. Динамичното заделяне на памет пък е нещо средно между двете – подобно на статичните данни тя се заделя върху част от оперативната (не-стек) памет, но се заделя по време на изпълнение и не се гарантира, че е занулена при създаването си, подобно на стековото заделяне. Освен малки печалби в бързина по време на изпълнение, смисъл за заделяне на статични данни има ако ни трябва голямо парче памет, което не искаме да заделяме динамично. Стекът обикновено е ограничен – по дефолт 1MiB за Windows и (вариращо) 2, 4 или 8 MiB за Linux – понякога можем да се сблъскаме със stack overflow – тоест мястото в стекът ни да „свърши". Този проблем може да се реши или като укажем на компилатора да задели по-голяма стекова рамка за приложението, или да менажираме по-добре паметта. Впрочем, stack overflow-ът е кофти проблем, тъй като е доста контра-интуитивен за хора, които не са се сблъсквали с него и също ужасен за дебъгване (за C/C++, в Java май е значително по-лесно). В следствие на тези си характеристики един от най-големите сайтове (по-скоро форуми) за програмистски въпроси е http://stackoverflow.com/
Връщайки се на темата за статичните данни и как не можем да заделяме твърде много информация в стека, почваме да се чудим къде тогава да я заделяме. Очевидно статичната памет е добър избор – тъй като е почти необятна, а също така и не пречи на performance-а. Решението е ясно – просто изнасяме всички масиви, които ни трябват извън функциите, иии... Не, това също не е хубава идея – грозно е от програмистска гледна точка (особено ако са много) – кодът става разхвърлян и труден за поддръжка и разбиране. Вместо това можем да заделяме статични данни от кода на функцията, в която се използват. Както казахме, те са заделени през цялото изпълнение на програмата, така че ще си ги имаме и няма да са в стека. Това става именно с кодовата дума static, която, зададена пред променлива, указва, че тя ще е статична. Това, че я заделяме като статична във функцията не значи, че тя е видима за останалите функции – тя все още е заделена в даден scope и е видима само в него. Предимство, обаче е, че ако

излезем от функцията, в която е била заделена, тя няма да се изтрие. Нещо повече, при следващо влизане в същата функция, паметта няма да е променена и ще можем да ползваме данните, които по-рано сме запазили в тази променлива.
В Java static се ползва подобно на C/C++ но тъй като там всичко е клас, една static променлива е обща за всички инстанции на класа. Прост пример защо би ни трябвало нещо такова е да направим брояч колко инстанции има от даден клас. Този брояч можем да го сложим като статична променлива в класа и да го увеличаваме при всяко викане на конструктор (и даже можем да го печатаме). При последователни създавания на инстанции от класа можете да видите, че бройката се увеличава.

Inline
Това е втората най-рядко срещаната от трите, тъй като модерните компилатори я ползват „невидимо" за нас дори да не присъства в кода ни. Каква е идеята: както знаете, извикването на функция заделя стекова рамка и извършва определена работа преди да почне да се изпълнява самата функция. Колкото и оптимизирани в това отношение да са модерните машини, при многократно викане на функцията (стотици хиляди, или милиони пъти) този „housekeeping" се натрупва и става усезаем. Housekeeping наричаме времето, отделено в работа, да се създаде функцията, да се задели памет за нейните аргументи, return code и други. Това е време, което е от нашата програма но не е за прякото изпълнение на нашия код. Друг пример за housekeeping е garbage collector-а в Java. Освен ако функцията не е рекурсивна, можем просто да пренесем кода на функцията там, където тя се вика – така няма да имаме това време и кодът ни ще е по-бърз. Това, обаче, е МНОГО грозен стил, тъй като в общия случай:

А) Функцията се вика на повече от едно място и ще имаме един и същ, повтарящ се код в проекта, което не е хубава практика;

Б) Една функция трябва да върши едно нещо, като такова „влагане" на функция в друга функция се обезкуражава в полза на разделяне на кода на отделни фрагменти.

Примери за функции, които биха могли да бъдат сравнително малки и ползвани на много места биха могли да бъдат примерно намиране на корен квадратен (sqrt()), вдигане на число на степен (pow()), намиране на разстояние между две точки, функция за превръщане от едно цветово пространство към друго (например YUV към RGB) и т.н. Както се досещате, тези функции са с по няколко реда, но биха могли да бъдат викани често (като последната би могла да бъде викана десетки милиони пъти в секунда при декодирането на филм, да кажем).
Стигнахме до конфликт на интереси – от една страна бихме желали кодът ни да е по-бърз като викането на функция бива избегнато, от друга не искаме да погрозняваме кода, като го направим. За това авторите на езика са въвели думата „inline", която, поставена пред функция, казва на компилатора да „пренесе" кода на функцията на мястото, където се вика. Това е сравнително лесно за компилатора да прецени сам – дали дадена функция може да бъде inline-ната или не, и има ли смисъл от това. Затова и повечето компилатори доста добре се справят с автоматично inline-ване. В повечето случаи можем да разчитаме, че са достатъчно умни да го правят вместо нас и съответно не е нужно да го слагаме в кода. Интересен факт е, че компилаторите до такава степен са умни в това отношение, че авторите им считат (може би правилно), че те биха били по-умни

дори от нас по отношение на inline-ване. Съответно, когато ние сложим пред дадена функция inline, това е по-скоро hint към компилатора да разгледа дали дадената функция не би била по-добре inline-ната – той ВЪПРЕКИ ВСИЧКО може да си реши да НЕ я inline-ва! За да го задължим да го прави са създадени отделни опции към компилатора (command-line опции), както и други кодови думи като __inline__ в GCC или __forceinline под Windows. Дори те обаче не гарантират 100%, че функцията ще е inline-ната.
Защо, обаче би му било на компилатора да НЕ inline-ва дадена функция? Нали казахме, че само печелим ако не я викаме ами я вкараме в другата функция (в машинния код)? Защо, всъщност, след написване на програмата, компилаторът не направи една ОГРОМНА функция и не вика само нея? Има няколко причини. Едната е, че не всяка функция може физически да бъде inline-ната. Например - как бихте inline-нали рекурсия? Съществуват и други причини за това, но няма да ги разглеждаме. Като цяло rule of thumb е ако времето за изпълнение на функцията е многократно по-голямо от няколкото милисекунди за "housekeeping" при извикването на функцията, тя да НЕ Е inline-ната. В общия случай ако функцията ви е по-голяма от няколко реда (примерно 3-5), то тя не е добър кандидат за inline.
В Java inlining-ът се извършва автоматично, но може да бъде прилаган само върху final методи, тъй като така се забранява override-ването на функцията от подклас. Тоест за да постигнете същото нещо трябва да направите метода final. Прост пример за това е даден в линковете горе.

Volatile
Третата запазена дума, която ще разгледаме, е най-рядко срещаната от трите. "volatile" на английски означава няколко неща, но значението, което нас ни интересува е „непостоянен, променлив, изменчив". Тъй като компилаторите правят много оптимизации върху нашия код без ние да знаем за това, понякога се налага да ги накараме да НЕ ГИ правят с цел да предотвратим проблем, за който ние знаем, но компилаторът няма как да знае. Например нека имаме следния код (хубав пример благодарение на Wikipedia):

```
static int foo;
void bar(void) {
    foo = 0;
    while (foo != 255);
}
```

Както сами виждате, веднъж извиквайки функцията bar() влизаме в безкраен цикъл (тъй като няма как променливата foo да се промени и условието foo != 255 ще бъде винаги true). Компилаторът вижда, че искаме да си направим безкраен цикъл и оптимизира кода до while(true); Това като цяло е оптимизация – пропуска се едно сравняване на всяка итерация от цикъла. И макар в общия случай това да е окей, ако имаме две нишки (thread-а), които ползват обща памет (включително тази променлива foo), то другата нишка може по някое време да промени стойността на променливата. Компилаторът, обаче, вече ще е сменил кода до while(true); и кодът ще си остане завинаги зациклил, дори променливата по някое време да бъде променена на 255.
За любознателните ще спомена, че когато искаме един thread да изчака друг thread да си свърши с работата и чак тогава първият да продължи,

имплементирано по този начин (с обща памет и цикъл, който проверява дали сме сменили даден флаг), се нарича „активно чакане". Тази техника е изключително проста за имплементация (и също така изключително неефективна и грозна), но поради простотата си има шанс да я срещнете на някои места (че дори и сами да я имплементирате (blush)).
Връщайки се на това, че компилаторът „оптимизира" кода по начин, по който ние не бихме желали, трябва да видим как можем да го накараме да не го прави. Предвиденият начин за това от създателите на езика е чрез кодовата дума "volatile". Тя „подсказва" на компилатора, че въпросната променлива (която е декларирана с volatile) е „непостоянна, променлива, изменчива", тоест може да бъде променена чрез начини, неизвестни за компилатора. Тогава той спира да извършва част от оптимизациите с нея, които разчитат на това, че нейната стойност не би могла да бъде променена ако не е променена в текущия код. Като цяло няма смисъл да я ползваме, освен ако нямаме някакъв вид shared memory (като например, както казахме, в многонишково приложение). Тъй като не всеки ден виждате многонишков код (особено пък във ФМИ) и съответно няма голям шанс да сте я виждали преди.
В Java тази кодова дума също съществува, но с известни разлики в смисъла. Когато тя е използвана за дадено поле (field) се гарантират следните неща:

А) (във всички версии на Java): Има глобална последователност от четения и писания във въпросната volatile променлива. Това означава, че всеки thread, който иска да access-не променливата ще прочете нейната стойност (а няма да се ползват кеширани нейни стойности).

Б) (от Java 5 насам): Четения и писания от volatile променлива създават happens-before relationship (това не съм много сигурен какво е, но пишат, че е подобно на заключване и отключване на mutex). Въпреки че volatile променлива е малко по-бърза от lock, все пак програмистите се съветват да използват истински lock в такива случаи, тъй като не са 100% еквивалентни.

Преди да завършим темата ще спомена как се измерва скоростта на процесора. Най-известният начин е в херци (мега, гига, там зависи от процесора), което на много места съм виждал (сравнително грешно) да се интерпретира като операции в секунда. Всъщност това са процесорни такта в секунда, тоест колко свои логически стъпки може да извърши той. Това не винаги (даже в повечето случаи не е) еквивалентно на колко операции в секунда може да извърши той. Главната причина за това е, че не всички операции изискват един такт. Някои от по-сложните могат да изискват 2, 3 или повече (дори до над 10 такта), докато най-простите (побитови операции, събирания) могат да се извършват по няколко на такт! Друга причина е, че различните типове изискват също различен брой такта. Например 64-битовия тип long long (long в Java) на 32 битова машина изисква двойно повече такта за операция, от стандартния за архитектурата (в случая 32 битов int). Нецелочислените операции също са по-бавни от целочислените. Като цяло при равен брой операции две програми могат да имат разлика в пъти в бързодействието си, в зависимост от това какви са операциите. Rule of thumb, което аз ползвам за да определя дали една операция е лека или тежка за процесора, е да помисля дали бих могъл сам да я напиша (и ако да - как). Разбира се знаем, че побитови операции, събирания и изваждания са леки (даже много леки) операции, но не можем да сме сигурни за, примерно, sin(), sqrt(),

log() и други подобни функции. Неточността на сравняването на скорости на процесорите в херци нараства с това, че различни процесори имплементират различни операции по различен начин и могат да изискват различен брой тактове. Прост пример: ако вземете един Pentium 4 и един Core i7 (и двата процесора са на Intel) на еднаква честота, от, да кажем, 3 гигахерца, разликата в производителността им би била убийствена. Затова, най-често за сравняване на процесори се използват програми, които тестват за колко време биха могли те да пресметнат последователност от операции (в общия случай разнообразна последователност от различни операции). Програмите, които правят това, се наричат benchmarks.
Като цяло можете да имате наум следните „тежести" на различните операции. Най-бързи са побитовите операции: and (&), or (|), xor (^), shift left (<<), shift right (>>), not (!), bitwise not (~). Съвсем малко по-бавни от тях (или със същата скорост, зависи от процесора) са събирането (+) и изваждането (-). Малко по-бавно от тях е умножението (*), доста по-бавни са делението (/) и намирането на остатък (%). След тях (доста по-бавни от div и mod) са различни операции, които са реализирани чрез редове: sqrt(), log() или чрез функции: pow(). Най-бавни са функции, които изискват системно прекъсване (тоест системни функции). Примери за такива са fprintf(), fscanf(), clock(), fork(), malloc() (operator new) и др.
Впечатляващо за мен беше как една програма може да бъде оптимизирана ДВОЙНО само като се смени "variable / 3.0" с "variable * 0.33333333".

Втори час
Задачи от интервюта