УВАЖАЕМЫЕ СТУДЕНТЫ! Изучите приведенную лекцию, законспектируйте основные тезисы и основные правила оптимизации.

Ответы на вопросы, фотоотчет, предоставить преподавателю на e-mail: r.bigangel@gmail.com до 06.02.2023.

При возникновении вопросов по приведенному материалу обращаться по следующему номеру телефона: (072)111-37-59, (Viber, WhatsApp), vk.com: https://vk.com/daykini

ВНИМАНИЕ!!! При отправке работы, не забывайте указывать ФИО студента, наименование дисциплины, дата проведения занятия (по расписанию).

Лекция 28

Тема: Эффективность и оптимизация программы.

Цель: Изучить основные вопросы связанные с оптимизацией программ

Это в наше-то время говорить об оптимизации программ? Бросьте! Не лучше ли сосредоточиться на изучении классов MFC или технологии .NET? Современные компьютеры так мощны, что даже Windows XP оказывается бессильна затормозить их!

Цель - определяет средства. Вот из этого, на протяжении всей книги, мы и будем исходить. Ко всем оптимизирующим алгоритмам будут предъявляется следующие жесткие требования:

- **оптимизация должна быть максимально машинно-независимой и переносимой на другие платформы (операционные системы) без дополнительных затрат и существенных потерь эффективности.** То есть никаких ассемблерных вставок! Мы должны оставаться исключительно в рамках целевого языка, причем, желательно использовать только стандартные средства, и любой ценой избегать специфичных расширений, имеющихся только в одной конкретной версии компилятора;
- **оптимизация не должна увеличивать трудоемкость разработки (в т. ч. и тестирования) приложения более чем на 10%-15%,** а в идеале, все критические алгоритмы желательно реализовать в виде отдельной библиотеки, использование которой не увеличивает трудоемкости разработки вообще;
- **оптимизирующий алгоритм должен давать выигрыш не менее чем на 20%-25% в скорости выполнения.** Приемы оптимизации, дающие выигрыш менее

20% в настоящей книге не рассматриваются вообще, т. к. в данном случае "овчинка выделки не стоит". Напротив, основной интерес представляют алгоритмы, увеличивающие производительность от двух до десяти (а то и более!) раз и при этом не требующие от программиста сколь ни будь значительных усилий. И такие алгоритмы, пускай это покажется удивительным, в природе все-таки есть!

- **оптимизация должна допускать безболезненное внесение изменений.** Достаточно многие техники оптимизации "умерщвляют" программу, поскольку даже незначительная модификация оптимизированного кода "срубает" всю оптимизацию на корню. И пускай все переменные аккуратно распределены по регистрам, пускай тщательно распараллелен микрокод и задействованы все функциональные устройства процессора, пускай скорость работы программы не увеличить и на такт, а ее размер не сократить и на байт! Все это не в силах компенсировать утрату гибкости и жизнеспособности программы. Поэтому, мы будем говорить о тех, и только тех приемах оптимизации, которые безболезненно переносят даже кардинальную перестройку структуры программы. Во всяком случае, грамотную перестройку. (Понятное дело, что "кривые" руки угробят что угодно - против лома нет приема).

Согласитесь, что такая постановка вопроса очень многое меняет. Теперь никто не сможет заявить, что, дескать, лучше прикупить более мощный процессор, чем тратить усилия на оптимизацию. Ключевой момент предлагаемой концепции состоит в том, что никаких усилий на оптимизацию тратить как раз не надо. Ну: почти не надо, - как минимум вам придется прочесть эту книжку, а это какие ни какие, а все-таки усилия. Другой вопрос, что данная книга предлагает более или менее универсальные и вполне законченные решения, не требующие индивидуальной подгонки под каждую решаемую задачу.

Назовем ряд правил оптимизации.

Правило I

Прежде, чем оптимизировать код, обязательно следует иметь надежно работающий не оптимизированный вариант

*("...прежде, чем класть все яйца в одну корзину - убедись, что ты построил действительно хорошую корзину").*аким образом прежде, чем приступать к оптимизации программы, убедись, что программа вообще-то работает.

Создание оптимизированного кода "на ходу", по мере написания программы, невозможно! Такова уж специфика планирования команд - внесение даже малейших изменений в алгоритм практически всегда оборачивается кардинальными переделками кода. Потому, приступайте к оптимизации только после тренировки на "кошках", - языке высокого уровня. Это поможет пояснить все неясности и "темные" места алгоритма. К тому же, при появлении ошибок в программе подозрение всегда падает именно на оптимизированные участки кода (оптимизированный код за редкими исключениями крайне ненагляден и чрезвычайно трудно читаем, потому его отладка - дело непростое), - вот тут-то и спасает "отлаженная кошка". Если после замены оптимизированного кода на не оптимизированный ошибки исчезнут, значит, и в самом деле виноват оптимизированный код. Ну, а если нет, то ищите их где-нибудь в другом месте.

Правило II

Помните, что основой прирост оптимизации дает не учет особенностей системы, а алгоритмическая оптимизация. Никакая, даже самая "ручная" оптимизация не позволит существенно увеличить эффективность пузырьковой сортировки или процедуры линейного поиска. Правильное планирование команд и прочие программистские трюки ускорят программу в лучшем случае в несколько раз. Переход к быстрой сортировке (quick sort) и двоичному поиску сократят время обработки данных как минимум на порядок, - как бы криво ни был написан программный код. Поэтому, если ваша программа выполняется слишком медленно, лучше поищите более эффективные математические алгоритмы, а не выжимайте из изначально плохого алгоритма скорость по капле.

Правило III

Не путайте оптимизацию кода и ассемблерную реализацию. Обнаружив профилировщиком узкие места в программе, не торопитесь переписывать их на ассемблер. Сначала убедитесь, что все возможное для увеличения быстродействия кода в рамках языка высокого уровня уже сделано. В частности, следует избавиться от прожорливых арифметических операций (особенно обращая внимание на

целочисленное деление и взятие остатка), свести к минимуму ветвления, развернуть циклы с малым количеством итераций: в крайнем случае, попробуйте сменить компилятор (как было показано выше - качество компиляторов очень разнится друг от друга). Если же и после этого вы останетесь недовольны результатом тогда...

Правило IV

Прежде, чем порываться переписывать программу на ассемблер, изучите ассемблерный листинг компилятора на предмет оценки его совершенства. Возможно, в неудовлетворительной производительности кода виноват не компилятор, а непосредственно сам процессор или подсистема памяти, например. Особенно это касается наукоемких приложений, жадных до математических расчетов и графических пакетов, нуждающихся в больших объемах памяти. Наивно думать, что перенос программы на ассемблер увеличит пропускную способность памяти или, скажем, заставит процессор вычислять синус угла быстрее. Получив ассемблерный листинг откомпилированной программы (для Microsoft Visual C++, например, это осуществляется посредством ключа /FA), бегло просмотрите его глазами на предмет поиска явных ляпов и откровенно глупых конструкций наподобие: MOV EAX, [EBX]\MOV [EBX], EAX. Обычно гораздо проще не писать ассемблерную реализацию с чистого листа, а вычищать уже сгенерированный компилятором код. Это требует гораздо меньше времени, а результат дает ничуть не худший.

Правило V

Если уж взялись писать на ассемблере, пишите максимально "красиво" и без излишнего трюкачества. Да, недокументированные возможности, нетрадиционные стили программирования, "черная магия", - все это безумно интересно и увлекательно, но: плохо переносимо, непонятно окружающим (в том числе и себе самому после возращения к исходному коду десятилетней давности) и вообще несет в себе массу проблем. Автор этих строк неоднократно обжигался на своих же собственных трюках, причем самое обидное, что трюки эти были вызваны отнюдь не "производственной необходимостью", а: ну, скажем так, "любовью к искусству". За любовь же, как известно, всегда приходится платить. Не повторяйте чужих ошибок! Не брезгуйте комментариями и непременно помещайте все ассемблерные функции в отдельный модуль. Никаких ассемблерных вставок - они практически непереносимы и создают очень много проблем при портировании приложений на другие платформы

или даже при переходе на другой компилятор. Единственная предметная область, не только оправдывающая, но, прямо скажем, провоцирующая ассемблерные извращения, это - защита программ, но это уже тема совсем другого разговора...

Профилировка программ

Профилировкой здесь и на протяжении всей книги мы будем называть измерение производительности как всей программы в целом, так и отдельных ее фрагментов, с целью нахождения "горячих" точек (Hot Spots), - тех участков программы, на выполнение которых расходуется наибольшее количество времени.

Согласно правилу "10/90", десять процентов кода "съедают" девяносто процентов производительности системы (равно как и десять процентов людей выпивают девяносто процентов всего пива). Если время, потраченное на выполнение каждой машинной инструкции, изобразить графически в порядке возрастания их линейных адресов, на полученной диаграмме мы обнаружим несколько высоченных пиков, горделиво возвышающихся над практически пустой равниной, усеянной множеством низеньких холмиков (пример показан далее на рисунке разд. " Практический сеанс профилировки с VTune в десяти шагах") Вот эти самые пики - "горячие" точки и есть.

Почему "температура" различных участков программы столь неодинакова? Причина в том, что подавляющее большинство вычислительных алгоритмов так или иначе сводятся к циклам, - т. е. многократным повторениям одного фрагмента кода, причем зачастую циклы обрабатываются не последовательно, а образуют более или менее глубокие иерархии, организованные по типу "матрешки". В результате, львиную долю всего времени выполнения, программа проводит в циклах с наибольшим уровнем вложения, и именно их оптимизация дает наилучший прирост производительности!

Громоздкие и тормозные, но редко вызываемые функции оптимизировать нет ни какой нужды, - это практически не увеличит быстродействия приложения (ну разве что только в том случае, если они совсем уж будут "криво" написаны).

Когда же алгоритм программы прост, а ее исходный текст свободно умещается в сотню-другую строк, то "горячие" точки нетрудно обнаружить и визуальным просмотром листинга. Но с увеличением объема кода это становится все сложнее и сложнее. В программе, состоящей из тысяч сложно взаимодействующих друг с другом функций (часть из которых это функции внешних библиотек и API -

Application Programming Interface, интерфейс прикладного программирования - операционной системы) далеко не так очевидно: какая же именно из них в наибольшей степени ответственна за низкую производительность приложения. Естественный выход - прибегнуть к помощи специализированных программных средств.

Профилировщик (так же называемый "профайлером") - основной инструмент оптимизатора программ. Оптимизация "в слепую" редко дает хороший результат. Помните пословицу "самый медленный верблюд определяет скорость каравана"? Программный код ведет себя полностью аналогичным образом, и производительность приложения определяется самым узким его участком. Бывает, что виновницей оказывается одна-единственная машинная инструкция (например, инструкция деления, многократно выполняющаяся в глубоко вложенном цикле). Программист, затратив воистину титанические усилия на улучшение остального кода, окажется премного удивлен, что производительность приложения едва ли возросла процентов на десять-пятнадцать.

Правило номер один: ликвидация не самых "горячих" точек программы, практически не увеличивает ее быстродействия. Действительно, сколько не подгоняй второго сзади верблюда - от этого караван быстрее идти не будет (случай, когда предпоследней верблюд тормозит последнего - это уже тема другого разговора, требующего глубоких знаний техники профилировки, а потому и не рассматриваемая в настоящей книге).

Цели и задачи профилировки

Основная цель профилировки - исследовать характер поведения приложения во всех его точках. Под "точкой" в зависимости от степени детализации может подразумеваться как отдельная машинная команда, так целая конструкция языка высокого уровня (например: функция, цикл или одна-единственная строка исходного текста).

Большинство современных профилировщиков поддерживают следующий набор базовых операций:

- определение общего времени исполнения каждой точки программы (total [spots] timing);

- определение удельного времени исполнения каждой точки программы ([spots] timing);
- определение причины и/или источника конфликтов и пенальти (penalty information);
- определение количества вызовов той или иной точки программы ([spots] count);
- определение степени покрытия программы ([spots] covering).

Общее время исполнения

Сведения о времени, которое приложение тратит на выполнение каждой точки программы, позволяют выявить его наиболее "горячие" участки. Правда, здесь необходимо сделать одно уточнение. Непосредственный замер покажет, что, по крайней мере, 99,99% всего времени выполнения профилируемая программа проводит внутри функции main, но ведь очевидно, что "горячей" является отнюдь не сама main, а вызываемые ею функции! Чтобы не вызывать у программистов недоумения, профилировщики обычно вычитают время, потраченное на выполнение дочерних функций, из общего времени выполнения каждой функции программы.

Рассмотрим, например, результат профилировки некоторого приложения профилировщиком profile.exe, входящего в комплект поставки компилятора Microsoft Visual C++.

Листинг 1.1. Пример профилировки приложения профилировщиком profile.exe

```
      Func        Func+Child         Hit
      Time  %       Time     %     Count  Function
---------------------------------------------------------
   350,192  95,9     360,982  98,9    10000 _do_pswd (pswd_x.obj)
     5,700   1,6       5,700   1,6    10000 _CalculateCRC (pswd_x.obj)
     5,090   1,4      10,790   3,0    10000 _CheckCRC (pswd_x.obj)
     2,841   0,8     363,824  99,6        1 _gen_pswd (pswd_x.obj)
     1,226   0,3     365,148 100,0        1 _main (pswd_x.obj)
     0,098   0,0       0,098   0,0        1 _print_dot (pswd_x.obj)
```