ЭФФЕКТИВНАЯ РАБОТА С ПАМЯТЬЮ ЭВМ

Лукьянов В. Н.
Кафедра вычислительных методов и программирования, Белорусский государственный университет информатики и радиоэлектороники
Минск, Республика Беларусь
E-mail: uladzislau.luk@gmail.com

В введении объясняется необходимость большого объема ОЗУ. В первом и втором разделах рассказывается о важности локальности, кэше и времени доступа к памяти ЭВМ. В третьем разделе предлагается альтернатива стандартному динамическому выделению памяти. Заключение.

Введение

Решаемые разработчиками задачи, по объемам обрабатываемых данных, можно разделить на два типа:

- задачи с большим количеством обрабатываемой информации;
- задачи с малым объемом обрабатываемых данных.

В первом случае, в следствии огромного объема данных, информация в адресном пространстве находится на значительном удалении друг от друга и крайне редко модифицируется, что затрудняет ее обработку процессором, ввиду низкой скорости операций запроса данных из оперативной памяти. Во втором случае объем обрабатываемой информации мал, что позволяет производить меньше обращений к ОЗУ и хранить данные в кэше процессора. В настоящее время количество задач требующих обработки малого объема информации постоянно снижается. Однако, следует помнить, что если стоит выбор между машиной с очень маленьким объемом ОЗУ, но с огромной ее скоростью и машиной с относительно большим объемом ОЗУ, вторая машина всегда выигрывает, если размер обрабатываемой информации превышает небольшой объем оперативной памяти первой машины.

I. Локальность

SRAM используется для создания временных копий данных в оперативной памяти, которые, с большой вероятностью, в скором времени будут использованы процессором. Это возможно, благодаря тому, что код программы и данные имеют временную и пространственную локальность. Это означает, что в течение короткого периода времени существует большая вероятность повторного использования одного и того же кода или данных. Для кода это означает, что в нем, скорее всего, есть циклы, так что один и тот же программный код будет выполняться снова и снова (идеальный вариант для пространственной локализации). Доступ к данным также в идеале ограничен небольшими регионами. Даже если память, используемая в течение короткого периода времени, не находится близко друг к другу, существует большая вероятность того, что одни и те же данные будут использованы повторно в течение длительного времени (временная локальность). Для кода это означает, например, что в цикле производится вызов функции и эта функция находится в другом месте адресного пространства. Функция может находиться на значительном удалении в памяти, но вызовы этой функции будут близки во времени. Для данных это означает, что общий объем используемой одновременно памяти (размер рабочего блока) в целом ограничен, но использованная память, в результате произвольного доступа к ней, находится далеко друг от друга. Понимание того, что локальность существует, является ключом к пониманию концепции кэша процессора.

II. Кэш

Вскоре после внедрения кэша система усложнилась. Разница в скорости между кэшем и основной памятью снова увеличилась, до такой степени, что был добавлен другой уровень кэша, больше и медленнее, нежели первый. Исключительное увеличение размера кэша первого уровня не было возможным по экономическим причинам. Уже сегодня существуют ЭВМ с тремя уровнями кэша (см. рис. 1).

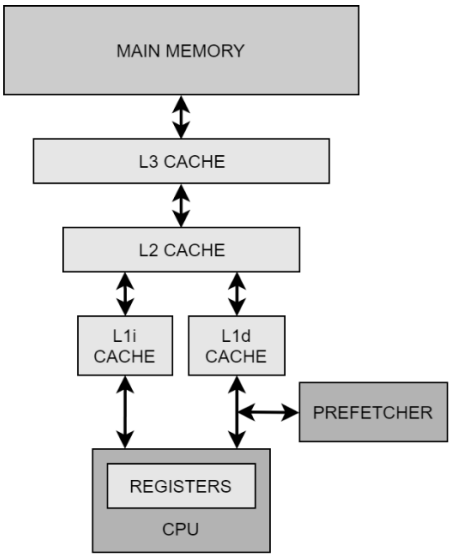

Рис. 1 – Схема ЦПУ с тремя уровнями кэша

Записи, хранящиеся в кэше, представляют собой не отдельные слова, а «строки» из множе-

ства последовательных слов. В ранних кэшах эти «строки», чаще их называют кэш-линиями, были 32 байта длиной; в настоящее время их размер составляет 64 байта. Если ширина шины памяти составляет 64 бита, то это означает, что на одну кэш линию приходится восемь операций передачи данных [1]. Оценочная стоимость во времени этой и других операций отражена в таблице 1.

Таблица 1 – Время выполнения операций с памятью

Операция	Время, нс
Доступ к кэшу данных первого уровня	0,5
Доступ к кэшу инструкций первого уровня	5
Доступ к кэшу второго уровня	7
Доступ к оперативной памяти	100
Последовательное считывание 1 Мбайта данных из памяти	250000

III. Распределитель памяти

Динамическое выделение памяти с помощью malloc() или глобального оператора new из C++ является крайне медленной операцией, ввиду дополнительных служебных операций на каждый вызов, использования кучи и фрагментации виртуальной памяти. Для предотвращения такого поведения рекомендуется использовать разработанные для конкретной цели распределители памяти. В системах с большим количеством операций считывания из памяти и записи в память, память, необходимая на протяжении всей жизни приложения, выделяется на этапе инициализации программного комплекса. Дальнейшие манипуляции с ней производятся через специальные распределители памяти, также называемые аллокаторами [2–3]. Рассмотрим основные типы аллокаторов:

1. Линейный аллокатор. Критически важным для этого аллокатора является то, что он не добавляет никаких дополнительных служебных операций при выделении и не изменяет предварительно выделенный буфер памяти. Не все распределители памяти будут обладать этими свойствами. Это делает его идеальным для низкоуровневых систем или для работы с памятью, доступной только для чтения;
2. Аллокатор, основанный на стеке. Чрезвычайно эффективен, когда необходимо выполнить несколько операций распределения памяти, и все они могут быть выполнены в обратном порядке. Кроме того, он также может быть использован для хранения данных, которые должны оставаться постоянными в течение некоторого времени с постоянно меняющимися данными в дополнение к ним;
3. Аллокатор, основанный на списке освобождения. Является «аварийным» аллокатором. Он предназначен для использования в качестве стратегии резервирования в ситуациях с малым объемом памяти, когда память недоступна для структур управления, необходимых другим распределителям памяти. В таких ситуациях аллокатор свободного списка гарантирует, что память не будет потеряна, но с несколькими недостатками:
   - сложность операций в свободном списке пропорциональна количеству свободных блоков;
   - структуры данных обладают низкой локальностью и, следовательно, потенциально низкой производительностью кэш-памяти.
4. Пул памяти. Выделяет область памяти один раз и разделяет эту область на фрагменты, которые точно соответствуют M объектам размером N. Пул памяти крайне полезен при выделения или освобождения памяти для объектов одного размера и с необходимостью постоянного динамического инстанциирования или уничтожения. Большинство из этих объектов создаются и уничтожаются в совершенно случайном порядке, в силу их динамической природы. Поэтому желательно иметь возможность выделять и освобождать память с минимальной фрагментарностью [4–5].

Заключение

Что стоит помнить и использовать в работе:

- временная и пространственная локальность крайне важна;
- чем больше данных хранится в кэше, тем быстрее они обработаются;
- чем ближе находятся данные, тем больше информации добавится в кэш за одну операцию чтения;
- стандартные способы динамического выделения памяти медленны;
- под каждую задачу существует соответствующий распределитель памяти.

Список литературы

1. What Every Programmer Should Know About Memory [Electronic resource] / U. Drepper. – Raleigh, N.C., 2007. – Mode of access: https://people.freebsd.org/∼lstewart/articles/. – Date of access: 01.10.2019.
2. Blunden, B. Memory management: algorithms and implementation in C/C++ / B. Blunden // Plano. Wordware Publishing, Inc., 2002. – 360 p.
3. Gregory, J. Game Engine Architecture / J. Gregory // Boca Raton. CRC Press, 2018. – 1240 p.
4. Building your own memory manager for C/C++ projects [Electronic resource] / A. Sen, R. Kardam. – Armonk, N.Y., 2008. – Mode of access: https://developer.ibm.com/tutorials/. – Date of access: 02.10.2019.
5. Jacob, B. Memory Systems: Cache, DRAM, Disk / B. Jacob, S. W. Ng, D. T. Wang // Burlington. Morgan Kaufmann, 2007. – 900 p.