суббота, 6 ноября 2010 г.

Thread-Safe Singleton

Singleton является, пожалуй одним из самых известных шаблонов проектирования, но на практике его традиционная реализация имеет множество проблем в многопоточной среде. В этой статье я попытаюсь объяснить, почему традиционные реализации singleton-а не потокобезопасны и какие способы существуют для достижения его потокобезопасной реализации.
Взглянем на два варианта традиционной реализации singleton-а.
Реализация #1 более известная как singleton Майерса.

```
1  class singleton
2  {
3  public:
4     static singleton* instance() {
5        static singleton inst;
6        return &inst;
7     }
8  private:
9     /* ... */
10 };
```

К сожалению, текущий стандарт C++ не содержит никаких упоминаний о потоках и в данном случае компилятор имеет полное право не генерировать потокобезопасный код (хотя некоторые компиляторы так поступают). Т.е. теоретически может быть создано более одного объекта, а параллельное выполнение пользовательского конструктора может вызвать и другие проблемы при обращении к общим ресурсам. В новом стандарте эта проблема будет устранена.

Реализация #2

```
1  class singleton {
2  public:
3     static singleton* instance() {
4        if (!inst)
5           inst = new singleton();
6        return inst;
7
8     }
9  private:
10    static singleton* inst;
11    /* ... */
12 };
```

Если singleton Майерса еще может работать на некоторых платформах и имеет потокобезопасное будущее в новом стандарте, то эта реализация не имеет никаких шансов на успех в многопоточной среде. Представьте ситуацию, когда два потока попытаются вызвать функцию instance() и первый поток останавливается, дойдя до 6-ой строки, передает управление второму потоку. Тот в свою очередь создает объект и возвращает указатель. После этого управление снова возвращается первому потоку, который еще раз создает объект и возвращает указатель.
Решить эту проблему довольно просто, надо лишь поставить lock_guard на входе функции.

```
1  class singleton {
2  public:
3     static singleton* instance() {
4        boost::lock_guard<boost::mutex> lk(mutex);
5        if (!inst)
6           inst = new singleton();
7        return inst;
8
```

```
9     }
10 private:
11    static singleton* inst;
12    static boost::mutex mutex;
13    /* ... */
14 };
```

Это полностью избавит нас от проблем, но это не самое эффективное решение. На самом деле блокирование mutex-ов довольно медленная операция и выполнять ее каждый раз несмотря на то, что необходима она только в первый (ведь когда объект уже существует, мы только возвращаем его указатель, а это безопасно) не очень-то правильно.
Широко известный шаблон DCLP (Double-Checked Locking Pattern) пытается решить эту проблему, однако известен он не тем, что так хорош, а тем, что с ним связано множество проблем и в мире программирования от него скорее больше вреда, чем пользы. Рассмотрим эти проблемы и попытаемся их исправить.

Double-Checked Locking Pattern

Реализация шаблона DCLP основывается на том, что блокирование потоков нужно только при создании объекта, в остальное время мы можем просто вернуть указатель на созданный объект. Идея хороша, но взглянем на реализацию.

```
1 singleton* singleton::instance() {
2    if (inst == 0) {
3       boost::lock_guard<boost::mutex> lk(mutex);
4       if (inst == 0) {
5          inst = new singleton();
6       }
7    }
8    return inst;
9 }
```

Сперва функция проверяет, не создан ли объект. Если объект не создан функция блокирует mutex и проверяет указатель на ноль еще раз. Вторая проверка необходима для того, чтобы удостовериться, что другой поток не создал объект в момент между первой проверкой и блокированием mutex-а. К сожалению этот pattern не работает в C++ и тому есть несколько причин.

Очередность выполнения

На самом деле в 5-ой строке предыдущего примера происходит три операции

- Выделение память для объекта
- Создание объекта в выделенной памяти
- Присваивание указателю inst адреса выделенной памяти

Но компилятору дается полное право на переупорядочивание этих операций. Он может поменять местами вторую и третью операцию, если это на его взгляд повысит производительность, ведь с точки зрения однопоточной программы (а стандарт C++ других не знает) ничего не изменится. В таком случае второй поток может вернуть указатель inst и начать работу с указателем, хотя первый еще не создал объекта. Конечно, мы можем попытаться решить эту проблему с помощью временной переменной

```
1 singleton* singleton::instance() {
2    if (inst == 0) {
3       boost::lock_guard<boost::mutex> lk(mutex);
4       if (inst == 0) {
5          singleton* temp = new singleton();
6          inst = temp;
7       }
8    }
9    return inst;
10 }
```

Но, к сожалению, проблему этот трюк не решает. Хороший оптимизатор сразу же вычислит, что переменная temp не нужна и заменит этот код предыдущим и тут не поможет ни вывод переменной temp в глобальную область видимости, ни даже в другую единицу трансляции. Удалять ненужные куски кода - задача оптимизатора и он свое дело знает. Даже если Вы нашли способ обмануть оптимизатор и написали код, который работает сегодня, со сменой компилятора (или даже с обновлением) он может перестать. А значит, для решения проблемы нужны стандартные средства.

Volatile

В 1970 году Гордон Белл (Gordon Bell) предложил концепцию MMIO (Memory-Mapped I/O), идея которой заключалась в том, что некая внешняя аппаратура должна перехватывать операции со специфическим адресом в памяти и преобразовывать их в I/O запросы. Это позволяло работать с портами как с обычной памятью через специфичный для этого порта адрес в памяти, но реализация этой замечательной идеи не была лишена проблем.
Например

```
1  unsigned int *p = get_address();
2  unsigned int a, b;
3  a = *p;
4  b = *p;
```

Если указатель p указывает на специфичный для какого-то порта адрес, то очередность чтения из p играет решающую роль, ведь если строки 3 и 4 поменять местами переменные a и b примут другие значения. Более того, оптимизатор может просто заменить вторую строку на “b = a”, устанавливая тем самым для обоих переменных одинаковое значение. Не лишена проблем с оптимизатором и запись по адресу

```
1  *p = a;
2  *p = b;
```

Оптимизатор может (и скорее всего он так и поступит) просто напросто избавиться от первой строки, так как с его точки зрения в ней нет необходимости (во второй строке значение все равно перезаписывается).
Для решения подобных проблем в стандарт C++ было введено ключевое слово volatile. Работа с volatile переменной может носить неизвестный компилятору смысл, все операции с volatile переменной должны быть выполнены с точностью и той очередностью, с которой они написаны в исходном коде. Грубо говоря, стандарт C++ запрещает компилятору проводить какую-либо оптимизацию с переменными типа volatile. Кажется это именно то, что нам нужно. Сделаем статический указатель на singleton volatile.

```
1  class singleton {
2  public:
3     static singleton* instance() {
4        if (inst == 0) {
5           boost::lock_guard<boost::mutex> lk(mutex);
6           if (inst == 0) {
7              singleton* temp = new singleton();
8              inst = temp;
9           }
10       }
11       return inst;
12    }
13 private:
14    static singleton* volatile inst;
15    static boost::mutex mutex;
16    int x;
17    singleton() {
18       x = 5;
19    }
20 };
```

После встраивания конструктора класса singleton код функции instance() может выглядеть вот так.

```
1  static singleton* instance() {
```

```
    if (inst == 0) {
       boost::lock_guard<boost::mutex> lk(mutex);
       if (inst == 0) {
          singleton* temp = static_cast<singleton*>(
             operator new(sizeof(singleton)));
          temp->x = 5;
          inst = temp;
       }
    }
    return inst;
 }
```

Несмотря на то, что переменная temp – volatile, *temp таковой не является (а значит и x тоже), а значит, компилятор волен перенести присваивание temp->x после присваивания inst = temp, что приведет к тому, что некий поток может начать использовать объект, несмотря на то, что его создание пока не завершено. Ну что ж, попробуем сделать volatile и указатель, и объект.

```
static volatile singleton* volatile inst;
```

К сожалению и это проблемы не решает, так как volatile объект становится только после создания, т.е. после того, как завершится работа его конструктора, а это приводит нас к первоначальной проблеме.
Мы можем попробовать изменить код конструктора

```
singleton() {
   static_cast<volatile int&>(x) = 5;
}
```

Теперь присваивание переменной x должно произойти до присваивания переменной inst, так как они оба volatile, но, к сожалению и это не решает всех возможных проблем.

Больше процессоров – больше проблем

Как известно в многопроцессорных/многоядерных SMP системах существует проблема синхронизации кэша (cache coherency problem). Подробнее об этой проблеме можно почитать в статье memory barriers. Это серьезное препятствие для реализации DCLP на многопроцессорных системах. Фактически один из потоков может увидеть ненулевой указатель еще до того, как первый завершит инициализацию объекта. Стандарт гарантирует очередность операций с volatile переменной, но он ничего не говорит о многопоточной среде. Это возвращает нас к первоначальной проблеме. Несмотря на то, что некоторые компиляторы генерируют барьеры памяти для volatile переменных, это решение не является переносимым. На самом деле переносимого и платформо-независимого решения стандарт C++ на сегодняшний день не предлагает и надо пользоваться специфичными для компилятора и платформы средствами.
Исправленный код функции instance() класса singleton будет выглядеть так

```
static volatile singleton* instance() {
   singleton* temp = inst;
   read_memory_barrier();
   if (inst == 0) {
      boost::lock_guard<boost::mutex> lk(mutex);
      if (inst == 0) {
         temp = new singleton();
         write_memory_barrier();
         inst = temp;
      }
   }
   return inst;
}
```

Стоит заметить, что барьеры избавляют нас от надобности использовать volatile, так как компилятор учитывает их при оптимизации.

Переносимое решение

Переносимой реализации DCLP с существующим стандартом C++ нет, но это не такая уж и большая проблема. Есть множество других решений.
Вместо того, чтобы каждый раз вызывать singleton::instance() пользователь может один раз сохранить указатель в какую-нибудь переменную и обращаться к объекту уже через него.

```
1  singleton* sptr = singleton::instance();
2  singletonPtr->func2();
3  singletonPtr->func3();
4  singletonPtr->func1();
```

Можно хранить по одному указателю для каждого отдельного потока в thread local storage (хотя текущий стандарт не предоставляет решения для TLS). Можно вообще избавиться от lazy инициализации и создавать объект из функции main() при входе.
Наилучшим решением, на мой взгляд, является boost::call_once, который и был создан для решения подобных проблем. Его реализация довольно эффективна и основана на interlocked функциях, использующих барьеры, к тому же call_once уже добавлен в черновик нового стандарта C++0x.
Все это естественно говорится с учетом того, что Вы обдумали архитектуру своего приложения и решили, что singleton Вам действительно необходим.

Ссылки

- C++ and the Perils of Double-Checked Locking
- boost online documentation
- C++ standard

Labels: барьеры памяти, пот
