Compact & Fast Token Allocator

Ted Nguyen, 11 Jun 2010 LGPL3

4.36 (7 votes)

Rate this:

Keep track of assigned integer tokens and ids with minimal overhead

Introduction

For over a decade, I have worked on various projects involving application middleware, class libraries, 3D Graphics SDK, or game engines. Invariably there is a need to keep track of assignments and reuse of tokens, handles, contexts id, etc.

The simplest solution is just to keep incrementing a counter whenever a new token is allocated. This solution does not allow for deallocating a token. When the counter wraps back to zero, there is a possibility of collision with previously allocated tokens. This scheme works fine if the application exits before the counter wraps around.

Another method is to keep a set or list of allocated tokens. When a token is deallocated, it gets moved to another list for reuse by the next allocation. There are many variations to this scheme; but the storage cost is ultimately one integer per token.

The method that requires the least storage is clearly a bitmask where the index of each bit represents one token. A bit is set to one to indicate it is allocated and clear to zero to indicate it is available. Complexity arises when searching for the first available (zero) bit. A naive implementation would search linearly through the bitmask. This article explores a storage scheme and an algorithm that can search through a bitmask in log32(n) time. For example, the first zero bit anywhere in a bitmask of one million bits can be found with only 4 comparisons.

Background

First, we need tools for manipulating bits. Specifically, we need the ability to find the first zero bit in an integer. I came across this website with really elegant bit manipulation algorithms.

```
// SWAR (SIMD Within A Register) algorithms for manipulating bits in an integer
// default implementation works best for 32-bit unsigned integer (unsigned long).
//  Specializations should be created to optimize other types
template <class T> size_t countOneBits(register T x)
{ // 32-bit recursive reduction using SWAR...
```

```
  x -= ((x >> 1) & 0x55555555);  // map 2-bit values into sum of 2 1-bit values
  x = (((x >> 2) & 0x33333333) + (x & 0x33333333));
  x = (((x >> 4) + x) & 0x0f0f0f0f);
  x += (x >> 8);
  x += (x >> 16);
  return(x & 0x0000003f);
}

template <class T> T foldBitsLeft(register T x)
{ // Replicate the lowest 'one' bit into higher positions
  x |= (x << 1);
  x |= (x << 2);
  x |= (x << 4);
  x |= (x << 8);
  return x | (x << 16);
}

template <class T> size_t lowestOneIdx(register T x)
{ return countOneBits(~foldBitsLeft(x)); }
```

A fellow member of this site by the name of waykohler

just pointed out a much faster and simpler method for finding the first one bit using assembly and intrinsics. If you use either gcc or msvc, you might want to try this instead of the SWAR functions above.

```
size_t lowestOneIdx(register unsigned long bits)
{
#ifdef __GNUC__
  if(bits) {
    return __builtin_ctz(bits);
  }
#elif defined(_MSC_VER)
  long idx;
  if(_BitScanForward(&idx,bits)) {
    return idx;
  }
#else
  #error "Unknown compiler"
#endif
  return 32;
}
```

Recursive Template Bitmask

A bitmask is usually implemented as an array of integers. SWAR algorithms above only work on individual integers. These algorithms must be extended to work on an array of integers without the need for linear search loops. This is where the magic of C++ recursive templates can help. We start out with a generic template for handling an array of integers.

```
typedef unsigned long word_type;
enum
{
  BITS_COUNT = 32,
  BITS_SHIFT = 5,                 // used in rounding shifts
  BITS_MASK = BITS_COUNT - 1,      // used in rounding masks
};
template <size_t Words>
class bit_array : protected bit_array<((Words+BITS_MASK) >> BITS_SHIFT)>
{
  typedef bit_array<((Words+BITS_MASK) >> BITS_SHIFT)>  base_type;
  word_type d_words[Words];     // storage of bits in this stage
public:
  bit_array()             { assign(false); }
  void assign(bool x)      { ::memset(this, x ? 0xff : 0x00, sizeof(bit_array)); }
  bool get(size_t n)  const { return 0 != d_words[n>>BITS_SHIFT] &
                          ((word_type)1 << (n&BITS_MASK)); }
  size_t find() const
  { // find first zero bit (template recursive )
    size_t i = base_type::find();
    return (i << BITS_SHIFT) +
        (i < Words ? lowestOneIdx(~d_words[i]) : 0);    // one branch here
  }
  void set(size_t n, bool x)
  { // set one bit at index n to x  (template recursive)
    size_t pos = n & BITS_MASK;       // bit position
    n >>= BITS_SHIFT;                // convert to index of word containing bit
    word_type& word = *(d_words + n);
    word &= ~((word_type)1 << pos);   // clear bit
    word |=   (word_type)x << pos;    // set bit if it is non-zero
    base_type::set(n, ~word == 0);    // update n bit in base
                                  // to one if word contains all ones
  }
};
```

Each template class handles an array of integers. It is entirely capable of setting or clearing any bit in its array. Its superclass is a bitmask where each bit is a flag corresponding to an integer in the array of the derived class. A flag of one indicates that the entire integer is allocated (filled with ones); a flag of zero indicates that an integer has at least one available (zero) bit. The derivation continues until the array size is one. We will need a template specialization for that.

```
template <> class bit_array<1>  // full specialization of the final stage
{
  word_type d_word;
public:
  bit_array() : d_word(0)   { }
  void assign(bool x)      { d_word = x ? ~(word_type)0 : 0; }
  bool get(size_t n)  const
```

```
        { return 0 != d_word & ((word_type)1 << (n&BITS_MASK)); }
    size_t find() const      { return lowestOneIdx(~d_word); }
    void set(size_t n, bool x)  // set one bit at index n to x
    {
      n &= BITS_MASK;                   // bit position
      d_word &= ~((word_type)1 << n); // clear bit
      d_word |=   (word_type)x << n;  // set bit if it is non-zero
    }
  };
```

To find an available (zero) bit in its array, each template class searches its superclass for the first available (zero) bit. This points to the exact integer that has an available (zero) bit. All that remains is to wrap the recursive template code into a class that is easier to use.

```
 /*!
    \class    FastBitMask
    \brief    An array of bits with fast search for the first zero bit
    \author   Ted Nguyen
  This class manages an array of bits in a hierarchy of masks that optimize
  the search for the first zero bit. This class uses template recursion to
  implement each level of the hierarchy. Each level holds an array of bits
  where a one indicates that a word in the derived level contains all ones.
  template parameters:
    max_bits : maximum number of bits managed by this array.
*/
template <size_t max_bits> class FastBitMask
{
protected:

  // code snippets above go here
  bit_array<((max_bits+BITS_MASK) >> BITS_SHIFT)>   d_array;
public:
  FastBitMask() : d_array()    { }
  explicit FastBitMask(bool x)  { d_array.assign(x); }
  void assign(bool x)             { d_array.assign(x); }
  static size_t max_size()         { return max_bits; }
  bool operator[](size_t n) const   { return get(n); }
  bool get(size_t n)  const        { return d_array.get(n); }
  size_t find() const
         { return d_array.find(); }  // find first zero bit
  void set(size_t n, bool x)
         { d_array.set(n, x); }     // set one bit at index n to x
};
```

The complete listing can be downloaded from here.

Using the Code

This algorithm provides all the required functionality of implementing a token allocator. A token allocator would contain an instance of class `FastBitMask<>`. The allocation method would check for the first available bit then setting that bit. The deallocation method would simply clear that bit.

```
  enum { COUNT = 1024 };
  FastBitMask<COUNT>      _bitbucket;   // bucket to keep track of available bits
public:
  int acquire()  // get next available index and mark it as used
  {
    size_t index = _bitbucket.find();   // find first zero bit in table
    if (_bitbucket.max_size() <= index) // out of range
      return -1;
    _bitbucket.set(index, 1);           // update bit in bucket
                                       // without any range checking
    return index;
  }
  bool release(int i_index)             // release specified index
                                       // and mark it as available
  {
    if (i_index < 0 || _bitbucket.max_size() <= i_index)
      return false;                    // index is outside valid range
    _bitbucket.set(i_index, 0);         // mark bucket as available
    return true;
  }
```

The size of the bitmask must be predetermined to keep runtime complexity at a minimum. I usually estimate the number of tokens required in the worst case scenario. Then I double or quadruple this quantity because the storage cost is low and the search cost is even lower.

This code is perfectly suited for applications that require allocation and deallocation of unique Ids. I have used this code to allocate Ids for OpenGL contexts. On another project, this code allocates unique identifiers for objects in a scenegraph. I am sure there are many uses for this algorithm. Drop a line and let me know if you have other novel uses. Thanks for reading.

License

Share

About the Author

Ted Nguyen

Software Developer (Senior)
United States
No Biography provided

```
size_t lowestOneIdx(register unsigned long bits)
{
#ifdef __GNUC__
  if(bits) {
    return __builtin_ctz(bits);
  }
#elif defined(_MSC_VER)
  long idx;
  if(_BitScanForward(&idx,bits)) {
    return idx;
  }
#else
  #error "Unknown compiler"
#endif
  return 32;
}
```