A C++ POOLED, SHARED MEMORY ALLOCATOR FOR THE STANDARD TEMPLATE LIBRARY

Marc Ronell
Electrical and Computer Engineering Department
University of Massachusetts
1 University Avenue, Lowell, Massachusetts 01854
marc_ronell@uml.edu

Abstract

A pooled, shared C++ allocator developed for use with the Standard Template Library (STL) is described. The allocator facilitates communication and control between multiple processes using data organized in STL container classes. The open source system has been compiled and tested on the Linux operating system.

1 Introduction

Research developing a computer architecture for an accelerated road traffic simulation machine [7, 6] yielded several important derivative research programs. The original research created an accelerated road traffic simulator which is designed to be fast enough to assist traffic management personnel in the event of an incident in an urban network. The simulator was determined to be at least 2 orders of magnitude faster than its software simulator counterparts. The hardware simulator research led to the development of two software products. The first was an open source road traffic simulator, Trafix, which was used for comparison with the hardware designs. The second more widely applicable result is the pooled, shared C++ allocator described in this article.

Trafix, an open source software simulator which incorporated and introduced the first allocator version, is developed as a two process system. One process is devoted to running a microscopic discrete event simulation. The second process serves as the basis for a future user interface to the simulator process. A natural elegant programming solution allows the user interface process to directly control objects in the simulator process. Both processes share a common set of Standard Template Library (STL) container classes which incorporate the pooled, shared allocator. The shared memory allocator allows the two processes to attach to the simulator objects directly. The user interface process can thereby manipulate and control the simulator. The first version of the pooled, shared memory allocator depends on initial global data structures being shared between a process and its forked descendant. The first process sharing the STL container classes forked the second process allowing both access to the shared global structures and memory segment mappings. Both the current and original allocator systems are available as open source from the Sourceforge website, http://allocator.sourceforge.net.

Potential users posted news forum requests for a system modification which would allow two or more unrelated processes to share the same STL container classes. The version described in this article was developed to serve that need. Two independent processes can separately open and share data stored in an STL container. The original code has been redesigned to simplify and reduce the size of the software. An added superclass allows users to open and map appropriate shared memory segments and to use semaphore locks to control access to the shared data structures. The system has adopted POSIX shared memory system calls to foster portability across UNIX platforms. The code was designed and tested on a GNU/Linux system.

The design takes advantage of several concepts recently added to the released version of the GCC compiler which make the shared memory allocator for the STL simpler. One new concept employed is the use of constant template parameters which, in the current allocator design, are used to both uniquely de-

fine the type of the allocator and to pass key values used to identify particular shared memory segments and address mappings. The second new advantage is the POSIX standard for shared memory which is now adopted by the Linux Kernel. Additional POSIX implementations are available in the kernel and GLIBC libraries. POSIX implementations, by design, facilitate portable shared memory implementations.
One of the most important aspects of this work is its leveraging of the well tested and dependable STL. Following Software Engineering concepts, the project adapts existing container classes enabling users to apply the STL containers as parallel processing communications and control functions elegantly.

2 Related Work

STL allocators and a description of their generic interface are available from several sources. Specifically, § 19.4 of [11] provides an outline of a pooled allocator class implementation. C++ allocators are defined in the C++ Standard [1] in two sections. § 20.1.5 describes the generic allocator interface. § 20.4.1 discusses the class std::allocator. In this paper, a class is an allocator if it conforms to the requirements of section § 20.1.5.
The standard boilerplate interface common to all STL allocators is well described in [2]. Given the existing literature which explains the interface, attention is devoted instead to the underlying memory allocation approach. Unlike the example in [2], the deallocate() method in this example needs both a pointer to the memory to be recycled as well as the element count. The underlying memory routines carefully track both the size and positions of the data locations. Released memory is recycled.

3 The Allocator Design

The allocator is designed to work with the Standard Template Library (STL) classes enabling multiple processes on the same machine to share containers and the objects which these containers hold.
STL allocators uncouple memory allocation from object creation [3]. Specifically, allocators encapsulate the low-level details of STL container memory management [2]. The STL container classes use allocator objects to provide memory for the data type of the objects held by the container. When the STL container class inserts a new object, the container requests memory from the allocator, and then copy constructs the requested object into the memory provided by the allocator. If the allocator provides shared memory, the container objects are visible to multiple processes. For consistent access, the sharing actually requires that both the container and the data objects stored within the container all be instantiated in shared memory.
The allocator both provides and disposes of memory for the STL container class data objects. The container class requests a number of empty object locations from the allocator. The allocator provides the requested space, returning a pointer to the first of the adjacently allocated locations. When the container is finished with the allocated memory, the container calls a deallocate() method to recycle the locations. The deallocate function takes a pointer to the start of the memory locations and the number of objects which describes the size of the memory to be recycled. The allocator releases the memory by adjusting its bit vector.

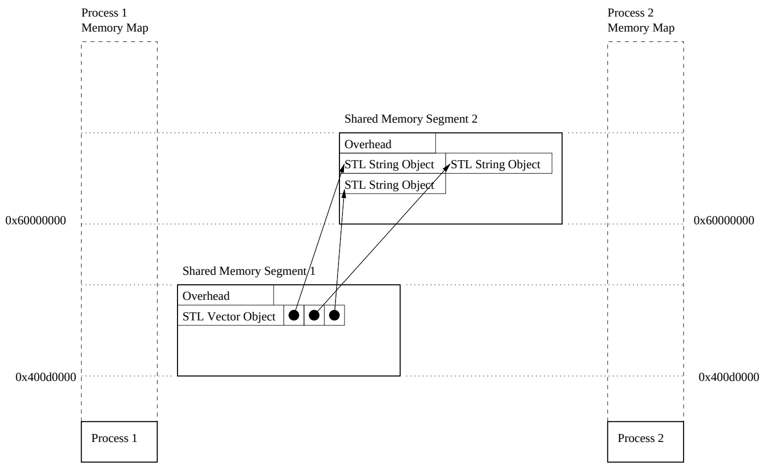

FIGURE 1: *The Allocator Overview*

The presented allocator design is composed of two basic layers. The bottom layer interfaces with the operating system's shared memory system calls. The shared memory handled in the lower software layers contains objects shared between the various attached processes. The lower layer therefore coordinates memory management between multiple processes. The upper layer of the allocator design interfaces to the STL container objects allocated by individual containers. The upper layer is concerned with memory management for each individual STL container, coordinating memory access on a per container basis.

One way in which this project differs from work such as the Hoard allocator project [4] is that this project specifically creates an allocator which is designed to allow STL container objects to be shared across multiple processes. Projects like Hoard concentrate on memory allocation speed.

3.1 The Allocator Structure

Figures presented in this paper follow the notation semantics used by Booch [5]. Classes are enclosed by dashed lines with the class name listed above a solid, horizontal line. Class attributes and methods are listed below the solid, horizontal line. Class relationships are identified by connecting lines. A connecting line sourcing from a full circle indicates that a class uses the attached class as a member attribute. A line emanating from an unfilled circle indicates that the attached class is used by the source class, for example, through a pointer. Twin slashes through a line on the source end indicate that the attached class is a private member of the source class. Similarly, twin, parallel lines next to an attribute or method within a class indicate private export control on the respective attribute or method. A single vertical line indicates protected export control. Attributes and methods without vertical lines are publicly exported. Inheritance is indicated by an arrowhead on a line which points to the superclass.

An example of memory allocation using the allocator class is illustrated in Figure 1. Two processes are shown. Process 1 has already created and initialized an STL vector object containing string elements using the allocator to place the vector in Shared Memory Segment 1, and a second allocator to place the string objects contained by the vector into Shared Memory Segment 2. Both shared memory segments are mapped to the same virtual memory address space in each process. Consistent address mapping allows the vector, which is instantiated in shared memory, to retrieve its elements in both processes. If process 1 establishes and fills the vector with its original three strings, process 2 can attach to the same shared memory segment, map it identically into process 2's virtual memory and then proceed to use the strings contained by the vector. Further, process 2 can add or modify the contents of the vector. Mutex variables are used to synchronize the process access to the vector object. In this example, separate shared memory allocators are used for the vector container and string data objects which the vector contains. For simplicity, both the allocator and its contained objects can be instantiated in the same shared memory segment.

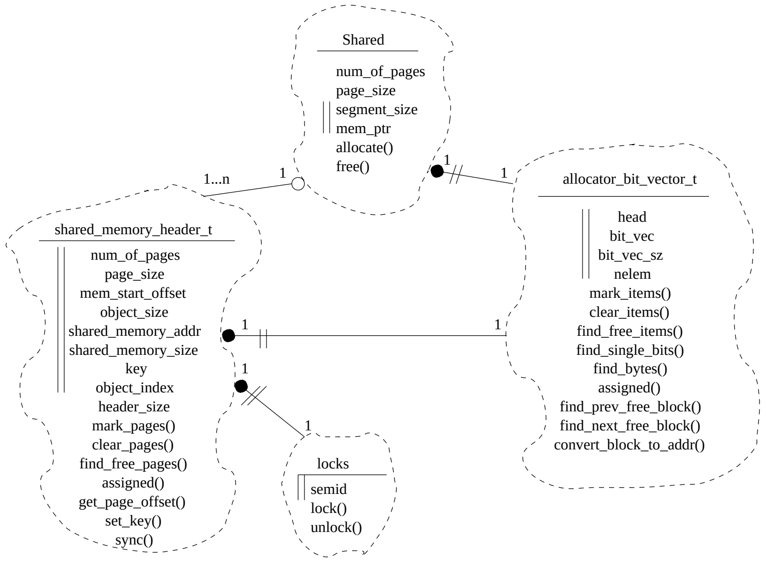

FIGURE 2: *The Allocator Classes which process the shared memory segments*

The shared memory segments are handled at the lowest level of the allocator design by the three classes illustrated in Figure 2. Higher level objects request the *shared* object to create new or open existing shared memory segments which serve as the source of available memory for the allocator. A *shared* object instantiates a *shared_memory_header_t* object at the beginning of new shared memory segments during their initialization. The *shared_memory_header_t* object fulfills requests from higher level classes for pages of memory from the shared memory segment. The header also uses a *locks* class allowing users to synchronize access to the STL containers instantiated in shared memory. Each shared memory segment is subdivided into pages, which are allocated to higher level classes and tracked using the *allocator_bit_vector_t* class. These four classes, illustrated in Figure 2, are used to process the shared memory segments.

The allocator top-level classes are illustrated in Figure 3. The *Pool_alloc* class consists of the generic std::allocator interface described by Table 32 of the C++ Standard [1]. *Pool_alloc* is the C++ allocator which can be used directly as a template parameter by the STL containers. The class calls a *Pool* object to request collections of memory locations where each location is the size of a data element stored by the STL container class. A *Pool* object requests memory segments from the low-level *shared* class as described in Figure 2. The Pool object parses the memory segments into *Chunk*s when the *Pool_alloc* object grows new memory. The *Chunk* class is a subdivision of a memory segment which tracks data element sized pieces of memory fulfilling STL container requests received by a *Pool_alloc* object. Whereas the shared memory segments are used by objects in multiple processes, class *Chunk* objects are assigned to individual allocators within a single process. One additional class, *Multi_Process_Pool_alloc* sits above the *Pool_alloc* and is described later. These four top-level allocator classes are illustrated in Figure 3.

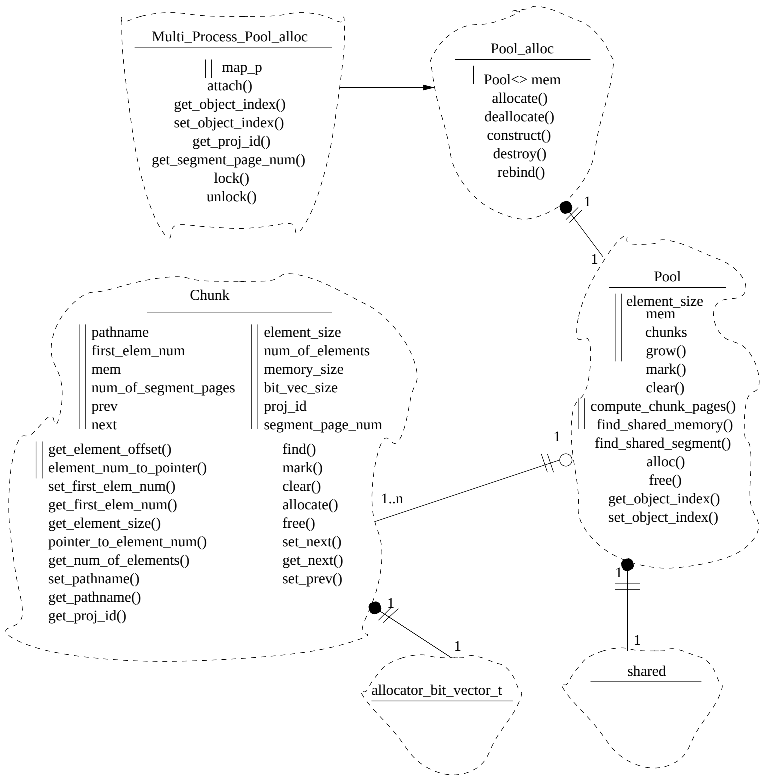

FIGURE 3: *Top-Level Allocator Class Structure*

Summarizing the relationship between the *shared* and *Pool* objects, the bottom level of the allocator consists of a *shared* object which is used by multiple processes. The *shared* object directly manipulates the shared memory segments. A *Pool* object sits above the *shared* object and works with an individual allocator for a container. A *Pool* object obtains its supply of memory from the *shared* object and amortizes the cost of obtaining that memory by requesting more than is needed and parceling out the extra for future requests.

3.2 The Shared Memory Subsystem

In this section, the pooled, shared memory allocator class structure is described from the bottom up. The bottom level classes, which handle the system shared memory calls directly, are illustrated in Figure 2. The *shared* object is used to create, attach, and destroy the shared memory segments. The original design used System V memory calls to access shared memory segments. With the current advances in GLIBC, the GNU C library used by the Linux operating system, the classes have been converted to work with the POSIX standard calls shm_open() and shm_unlink() [8, 9]. *shared* uses two method calls to create/attach and destroy shared memory segments. Method *allocate()* is used to create or attach to the segments and *free()* is used to release the memory.

When opening a shared memory segment, the class first attempts to open the segment as if the segment already exists. The key defining the particular shared memory segment is passed to the *shared* class as a constant template parameter. If the open fails, the class assumes that the shared segment does not yet exist, and creates the segment for the first time, flagging the new segment for initialization.

Once opened, the segment is truncated to the standard segment size using the *ftruncate()* system call. After being opened and sized, the shared memory segment is next mapped into the process virtual memory space using *mmap()*. The memory segment is mapped for both reading and writing, and is configured to be shared. The shared segments are mapped into each process's virtual memory at the location specified by a constant template parameter value. Mapping the memory to the same locations in all processes allows the STL containers in different processes to access their elements consistently. For shared segments which need to be created as opposed to just opened, the *shared* class instantiates a *shared_memory_header_t* object at the beginning of the segment using *placement*.

Placement is a C++ mechanism which instantiates an object at a specific memory location. For example, if `new (p)  T(a,b)` is invoked, T's constructor with parameters a and b is called creating an object of type T at the memory location pointed to by p. The object's destructor can be called without releasing any memory by `p->~T()`. The *shared* class uses placement to instantiate a copy of the *shared_memory_header_t* at the beginning of the shared memory segment. A process which subsequently attaches to the existing shared memory segment knows to attach to the object which contains coordination information.

The shared class method *free()* is used to release the shared memory segment with the POSIX call, *shm_unlink()*.

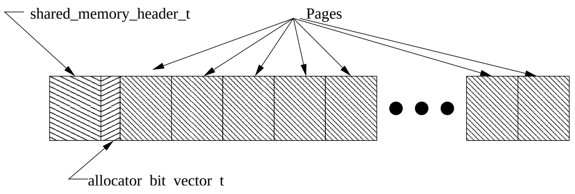

FIGURE 4: *The Shared Memory Segment Structure*

Once created or opened, the shared memory segments are handled by their embedded header classes, which are instantiated at the beginning of the shared memory segment. The header is followed by a bit vector which is used to keep track of unused, available pages in the segment. The allocatable memory space, which is provided to higher level objects upon request, starts after the bit vector and continues to the end of the shared segment. An illustration of the shared memory segment structure is shown in Figure 4.

The *shared_memory_header_t* stores information used by all processes accessing the segment. The header keeps track of which memory from the segment is available and which has been parceled out to the various processes. Keys used to create the segment uniquely identify it and are stored in the header along with an index to a map which can be used to store pointers logging currently active objects. The header's main function is to locate requested pages of memory and allocate them, or recycle returned pages of memory for reuse.

3.3 Pooled Allocator Subsystem

The Pooled Allocator Subsystem is a shared memory allocator created to work with the C++ STL. The allocator uses the Shared Memory classes described in Section 3.2 and dispenses the shared memory in element sized portions directly to the STL container classes. Once the container classes instantiate their objects using shared memory, the objects are visible to any subscribing process and can be accessed through the STL container object. Attaching processes need only define identical container typedefs and attach using the same shared memory key and address values. Both the key and the address are kept in external char array constants which are used as constant template parameters. These values are passed down to the *shared* class and serve as keys when opening and mapping the shared memory segment into process virtual memory. The Pooled Allocator Subsystem is composed of basically the three classes *Pool_alloc*, *Pool*, and *Chunk* which are illustrated in Figure 3. A fourth class, *Multi_Process_Pool_alloc* is used to assist processes in attaching to existing STL container objects instantiated in shared memory. This section describes the operation of these four top level classes in detail.

The lowest level of the Pooled Allocator Subsystem is the *Pool* class. The *Pool* class stores its collection of element sized memory in a list of *Chunk* objects. Each *Chunk* object contains a quantity of element sized memory locations which are provided directly to the allocator. When the memory contained within the Chunks list is depleted, the next attempt to *find()* available memory triggers a call to *grow()* which then attempts to replenish the depleted supply of memory. Method *grow()* is used to create a new *Chunk* object and insert the new chunk into the Chunk list managed by Pool. The *grow()* method requests its memory from the Pool's *shared* object attribute.

The *Pool* method, *grow()*, requests each subsequent *Chunk* object to be exponentially larger than the previously requested *Chunk* size. The requested chunk size may span multiple shared memory segment pages. The requests are made through *shared_memory_header_t* objects which search their segments to satisfy each memory request. Once the free pages of memory are located in the shared segment, they are marked as in use. Placement is used to instantiate a Chunk object at the beginning of the shared page set. The newly initialized Chunk is then inserted into the Pool's list of chunks.

The *Pool* object provides and recycles memory for the allocator through its methods *alloc()* and *free()*. The *alloc()* method searches through a Pool's Chunk list for a chunk containing enough memory to satisfy the allocator's request. If memory is not found, the function calls *grow()*, or upon failure by *grow()* throws an exception. When a chunk containing the required memory is found, the memory is marked as in use and a pointer to the memory is retrieved from the enclosing Chunk. The *free()* method is basically the reverse of the *alloc()* method. The *free()* method receives a pointer to the memory to be recycled along with a count of the sequential elements which are to be recycled starting at that pointer location. The method locates the appropriate chunk which contains the memory and calls the chunk to mark the memory as now available.

The *Chunk* objects are very similar to the *shared_memory_header_t* objects in that they both allocate and parcel out memory. Chunks, however, only work within one allocator object and the memory units they handle are element sized. Shared memory segments may be used and shared by multiple processes. Unlike the consistent shared memory segment size, the Chunk sizes themselves also vary. Their sizes increase exponentially as more chunks are required by a *Pool* object. Although the overall Chunk size may vary, the memory the *Chunk* allocates to the *Pool* class is always a constant data element size. Chunks are stored in a doubly linked list. The memory layout of a Chunk is illustrated in Figure 5. Like the shared segment layout, the *Chunk* starts with an instantiated *Chunk* object. The *Chunk* object is followed by an *allocator_bit_vector_t* object which is used to keep track of the Chunk's available element sized cells. After the bit vector, the rest of the chunk consists of data element sized memory units for allocation to the STL container classes.

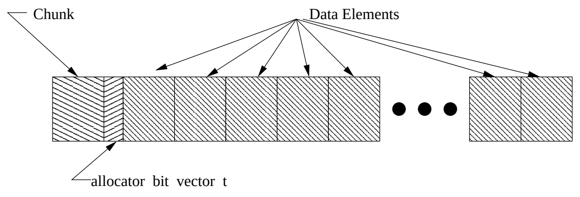

FIGURE 5: *The Chunk Structure*

The *Pool_alloc* class attributes and methods follow the Standard definition. The one exception is an added private *Pool* class attribute which is used to allocate and deallocate memory. *Pool_alloc* is the shared memory allocator. The attributes and methods of the standard interface are well documented by [2, 3, 10, 11] and the reader is referred to these sources for the interface description. The *Pool_alloc* class is designed to work with the STL container classes. One important difference between the allocator described in some of the earlier related work publications and the allocator presented here is the number of additional template parameters required by this new allocator. In addition to the element type, two other constant parameters are passed into the allocator. The two template parameters, *alloc_key*, and *alloc_addr* are shown in Table 1. The *alloc_key* parameter is used to identify which shared memory segment the allocator should open. The *alloc_addr* constant parameter defines the starting base address where the shared memory segment will be mapped in the process's virtual memory.

The final class used as part of the Pooled Allocator Subsystem is the *Multi_Process_Pool_alloc* class. This class is a subclass of *Pool_alloc*. Users create *Multi_Process_Pool_alloc* objects which in turn create instantiated versions of the STL containers in shared memory. These instantiated container objects are maintained in an index which is stored in shared memory. The index consists of an STL map object which uses *std::basic_string* as the key and a data object which holds a pointer and offset information to the container class. Other processes use their own version of *Multi_Process_Pool_alloc* to attach to the STL container class in shared memory.

```
template<class T,
         mem_space::allocator_key_t alloc_key,
         mem_space::allocator_addr_t alloc_addr=0>
class Pool_alloc {

};
```

Table 1: *The Pool_alloc Template Parameters*

4 Conclusion

The first version of the shared memory allocator required the processes using the allocator to share the same global data structures. The approach shared the container objects between the parent process and its children. Although this approach satisfied early requirements, a pooled, shared allocator which is able to share STL container class objects across unrelated processes is more widely applicable and was requested by the user community. This paper describes part of the effort to satisfy these user requests.

The current version of the pooled, shared memory allocator has been coded and successfully tested with unrelated processes. The approach allows the generic STL container classes to organize shared data structures. Table 2 illustrates code used to instantiate and populate a shared STL vector class. Three typedefs are used to define the class types required to instantiate the shared vector class. The first typedef, *my_vector_alloc_t* type, defines an allocator type which provides shared, pooled storage for integer data elements. The second template parameter, *key_val*, defines the specific shared memory segment to create or open. The last template parameter, *key_addr*, defines the address in the process's virtual address space where the shared memory segment must be mapped using the *mmap()* system call.

Once the shared, pooled allocator type is defined, the actual vector class, *my_vector_t* type can be defined. The second typedef, *my_vector_t*, uses the allocator defined by the first typedef, *my_vector_alloc_t*, creating a vector container type. This type uses the *Pool_alloc* allocator described in Figure 3 to generate integer data element storage locations in shared memory. Finally, the *Multi_Process_Pool_alloc* class can be defined. The *Multi_Process_Pool_alloc* object is used to create the shared memory segment, map the segment into the process virtual address space, instantiate the container class in shared memory and then attach to the instantiated container. The *Multi_Process_Pool_alloc* type specifies a fourth constant template parameter, *key_container*, which is used by secondary processes to gain access to container objects, in this case, retrieving a pointer to the vector object instantiate in shared memory. The same class is used to synchronize access to the data vector between attaching processes with mutex locks and to clean up and release the shared memory segments and semaphores when finished.

The code used to connect to the shared memory allocated STL container is similar to the code used to create the container. An example used to connect to the shared vector is provided in Table 3. In this example, the code starts with the same three typedefs used in Table 2. The external char array keys have identical values allowing the connecting code to attach to the same shared memory segment and map the segment onto the process's corresponding virtual address space. Once the connecting code has attached and mapped the shared memory using the *Multi_Process_Pool_alloc* object, the attach() method, assuming the vector has already been instantiated by the code of Table 3, provides a pointer to the container mapped to the *key_container* value. For example, a pointer to the vector holding the integers is returned. The connecting class can then read, modify, add, and delete from the contents of the shared memory vector. Mutex locks are used to synchronize process access to the shared vector's contents.

The shared, pooled memory class is proving to be a convenient and well organized approach to shared memory storage. The resulting allocator is open source and available from its website, http://allocator.sourceforge.net. Additional work profiling the code, reducing throughput bottlenecks by adding coding optimizations, and generally studying and reviewing the performance of the existing design need to be undertaken to improve the current system. Future work will integrate POSIX semaphores as they become available to Linux. Another target feature which follows intuitively from the current work is an STL allocator which can be used to coordinate shared memory between multiple machines as opposed to multiple processes all running on the same machine. This added feature requires an effective memory coherence design. The feedback and encouragement provided by the open source user community for this project is extremely valuable in determining its direction and success. Feedback can be provided through the project's web page forums.

References

[1] Accredited Standards Committee. *Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++*, December 1996. 2, 4

[2] Matthew H. Austern. The standard librarian: What are allocators good for? *C/C++ Users Journal*, December 2000. 2, 6

[3] Matthew H. Austern. A debugging allocator. *C/C++ Users Journal*, December 2001. 2, 6

[4] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multi-threaded applications. *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000. Cambridge, MA. 3

[5] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison Wesley, February 1994. ISBN: 0-201-89551-X. 3

[6] Marc Bumble, Lee Coraor, and Lily Elefteriadou. Exploring CORSIM runtime characteristics: Profiling a traffic simulator. *33rd Annual Simulation Symposium 2000 (SS 2000)*, pages 139–146, April 2000. 1

[7] Marc D Bumble and Lee Coraor. An architecture for a non-deterministic distributed simulator. *IEEE Transactions on Vehicular Technology*, 51(3):453–471, May 2002. 1

[8] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1990. ISBN 0-13-949876-1. 5

[9] W. Richard Stevens. *UNIX Network Programming*, volume 1. Prentice Hall, Inc., second edition, 1998. ISBN 0-13-490012-X. 5

[10] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1991. 6

[11] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

2, 6

```
#include <iostream>
#include <pooled_allocator.h>
#include <vector>

char key_val[] = "/allocate_key";  // needs a leading slash, see manpage.
char key_container[] = "container_lookup_key";
char key_addr[] = "0x400d0000";

int main(int argc, char** argv) {

  typedef pooled_allocator::Pool_alloc<int,
    key_val,
    key_addr> my_vector_alloc_t;

  typedef std::vector<int,my_vector_alloc_t> my_vector_t;

  pooled_allocator::Multi_Process_Pool_alloc<my_vector_t,
    int,
    key_val,
    key_container,
    key_addr> temp_alloc;

  my_vector_t* vec_ptr = temp_alloc.attach();
  const int& segment_num = temp_alloc.get_proj_id();
  const int& segment_page_num = temp_alloc.get_segment_page_num();
  temp_alloc.lock(segment_num,segment_page_num);
  vec_ptr->push_back(5);
  vec_ptr->push_back(6);
  vec_ptr->push_back(7);
  vec_ptr->push_back(8);
  temp_alloc.unlock(segment_num,segment_page_num);

  for (my_vector_t::iterator it = vec_ptr->begin();
       it != vec_ptr->end(); it++)
    std::cout << "Vector val: " << *it << std::endl;

  temp_alloc.shutdown();

  return 0;
}
```

Table 2: *STL vector example using the pooled, shared memory allocator*

```
#include <iostream>
#include <pooled_allocator.h>
#include <vector>

char key_val[] = "/allocate_key";  // needs a leading slash, see manpage.
char key_container[] = "container_lookup_key";
char key_addr[] = "0x400d0000";

int main(int argc, char** argv) {

  typedef pooled_allocator::Pool_alloc<int,
    key_val,
    key_addr> my_vector_alloc_t;

  typedef std::vector<int,my_vector_alloc_t> my_vector_t;

  pooled_allocator::Multi_Process_Pool_alloc<my_vector_t,
    int,
    key_val,
    key_container,
    key_addr> temp_alloc;

  my_vector_t* vec_ptr = temp_alloc.attach();
  for (my_vector_t::iterator it = vec_ptr->begin();
       it != vec_ptr->end(); it++)
    std::cout << "Vector val: " << *it << std::endl;
  const int& segment_num = temp_alloc.get_proj_id();
  const int& segment_page_num = temp_alloc.get_segment_page_num();
  temp_alloc.lock(segment_num,segment_page_num);
  vec_ptr->push_back(11);
  vec_ptr->push_back(12);
  vec_ptr->push_back(13);
  vec_ptr->push_back(14);
  temp_alloc.unlock(segment_num,segment_page_num);

  return 0;
}
```

Table 3: *Code example used to attach to shared STL vector*