Creating STL Containers in Shared Memory

By Grum Ketema, April 01, 2003

Shared memory is one of the IPC (interprocess communication) facilities available in every major version of Unix. It allows two or more processes to map the same set of physical-memory segments to their address space. Since the memory segments are common to all processes that are attached to them, the processes can communicate through the common data in the shared-memory segments. Thus, as the name implies, shared memory is a set of physical-memory segments that are shared among processes. When a process attaches to shared memory, it receives a pointer to the beginning of the shared segments; the process then can use the memory just like any other memory. Of course, care must be taken when a shared-memory segment is accessed or written to synchronize with another process that has access to the same physical memory.

Consider the following code, which works on most Unix systems:

```
//Get shared memory id
//shared memory key
const key_t ipckey = 24568;
//shared memory permission; can be
//read and written by anybody
const int perm = 0666;
//shared memory segment size
size_t shmSize = 4096;
//Create shared memory if not
//already created with specified
//permission
int shmId = shmget
  (ipckey,shmSize,IPC_CREAT|perm);
```

```
if (shmId ==-1) {
  //Error
}

//Attach the shared memory segment

void* shmPtr = shmat(shmId,NULL,0);

struct commonData* dp =
  (struct commonData*)shmPtr;

//detach shared memory
shmdt(shmPtr);
```

Types of Data Structures in Shared Memory

Care must be taken when placing data in shared memory. Consider the following structure:

```
Struct commonData {
  int sharedInt;
  float  sharedFloat;
  char* name;
Struct CommonData* next;
};
```

Process A does the following:

```
//Attach shared memory
struct commonData* dp =
  (struct commonData*) shmat
    (shmId,NULL,0);

dp->sharedInt = 5;
.
.
dp->name = new char [20];
strcpy(dp->name,"My Name");
```

```
dp->next = new struct commonData();
```

Some time later, process B does the following:

```
struct commonData* dp =
  (struct commonData*) shmat
    (shmId,NULL,0);

//count = 5;
int count = dp->sharedInt;
//problem
printf("name = [%s]\n",dp->name);
dp = dp->next;  //problem
```

Data members **name** and **next** of **commonData** are allocated from the heap in process A's address space. **name** and **next** are pointing to an area of memory that is only accessible by process A. When process B accesses **dp->name** or **dp->next**, it will cause a memory violation since it is accessing a memory area outside of its address space. At the minimum, process B will get garbage for the **name** and **next** values. Thus all pointers in shared memory should point to locations within shared memory. (That is why C++ classes that contain virtual function tables -- those that inherit from classes that have virtual member functions cannot be placed in shared memory -- is another topic.)

As a result of these restrictions, data structures designed to be used in shared memory usually tend to be simple.

C++ STL Containers in Shared Memory

Imagine placing STL containers, such as maps, vectors, lists, etc., in shared memory. Placing such powerful generic data structures in shared memory equips processes using shared memory for IPC with a powerful tool. No

special data structures need to be designed and developed for communication through shared memory. In addition, the full range of STL's flexibility can be used as an IPC mechanism. STL containers manage their own memory under the covers. When an item is inserted into an STL list, the list container automatically allocates memory for internal data structures to hold the inserted item.

Consider placing an STL container in shared memory. The container itself allocates its internal data structure. It is an impossible task to construct an STL container on the heap, copy the container into shared memory, and guarantee that all the container's internal memory is pointing to the shared-memory area.

Process A does the following:

```
//Attach to shared memory
void* rp = (void*)shmat(shmId,NULL,0);
//Construct the vector in shared
//memory using placement new
vector<int>* vpInA = new(rp) vector<int>*;
//The vector is allocating internal data
//from the heap in process A's address
//space to hold the integer value
(*vpInA)[0] = 22;
```

Process B does the following:

```
vector<int>* vpInB =
  (vector<int>*) shmat(shmId,NULL,0);

//problem - the vector contains internal
//pointers allocated in process A's address
//space and are invalid here
int i = *(vpInB)[0];
```

C++ STL Allocators to the Rescue

One of the type arguments to an STL container template is an allocator class. The allocator class is an abstraction of a memory-allocation model. The default allocator allocates memory from the heap. The following is a partial definition of the **vector** class in STL:

```
template<class T, class A = allocator<T> >
  class vector {
    //other stuff
};
```

Consider the following declaration:

```
//User supplied allocator myAlloc
vector<int,myAlloc<int> > alocV;
```

Assume **myAlloc<int>** allocates memory from shared memory. The vector **alocV** is constructed entirely from shared-memory space.

Process A does the following:

```
//Attach to shared memory
void* rp = (void*)shmat(shmId,NULL,0);
//Construct the vector in shared memory
//using placement new
vector<int>* vpInA =
  new(rp) vector<int,myAlloc<int>>*;
//The vector uses myAlloc<int> to allocate
//memory for its internal data structure
//from shared memory
(*v)[0] = 22;
```

Process B does the following:

```
vector<int>* vpInB =
```

```
  (vector<int,myAlloc<int> >*) shmat
    (shmId,NULL,0);

//Okay since all of the vector is
//in shared memory
int i = *(vpInB)[0];
```

All processes attached to the shared memory may use the vector safely. In this case, all memory allocated to support the class is allocated from the shared memory, which is accessible to all the processes. By supplying a user-defined allocator, an STL container can be placed in shared memory safely.

A Shared Memory Based STL Allocator

Listing 1 shows an implementation of the C++ Standard STL allocator. The STL allocator is itself a template. The **Pool** class does the shared-memory allocation and deallocation.

Listing 2 shows the **Pool** class definition. **Pool**'s static member **shm_** is of type **shmPool**. There is a single instance of **shmPool** per process, and it represents shared memory. **shmPool**'s constructor creates and attaches the desired size of shared memory. Shared-memory parameters, such as the shared-memory key, number of shared-memory segments, and size of each segment, are passed to the **shmPool** class constructor through environmental variables. The data member **segs_** is the number of shared-memory segments; **segSize_** is the size of each shared-memory segment. The **path_** and **key_** data members are used to create a unique **ipckey**. **shmPool** creates one semaphore for each shared segment to synchronize memory-management activities among processes attached to the shared-memory segment. **shmPool** constructs a **Chunk** class in each of the shared-memory segments. **Chunk** represents a shared-memory segment. For each shared-memory segment, the shared-memory identifier **shmId_**, a

semaphore **semId_** to control access to the segment, and a pointer to the **Link** structure that represents the free list are kept in the **Chunk** class.

Placing an STL Container in Shared Memory

Suppose process A places several STL containers in shared memory. How does process B find these containers in shared memory? One way is for process A to place the containers at fixed offsets in the shared memory. Process B then can go to the specified addresses to obtain the containers. A cleaner way is for process A to create an STL map in shared memory at a known address. Then process A can create containers anywhere in shared memory and store the pointers to the containers in the map using a name as a key to the map. Process B knows how to get to the map since it is at an agreed location in shared memory. Once process B obtains the map, it can use the containers' names to get the containers. Listing 3 shows a container factory. The **Pool** class method's **setContainer** places the map at a well-known address. The **getContainer** method returns the map. The factory's methods are used to create, retrieve, and remove containers from shared memory. The container types passed to the container factory should have **SharedAllocator** as their allocator.

Conclusion

The scheme described here can be used to create STL containers in shared memory. The total size of shared memory (**segs_* segSize_**) should be large enough to accommodate the STL container's largest size, since **Pool** does not create new shared memory if it runs out of space.

The complete source code is available for download at <www.cuj.com/code>.

References

Bjarne Stroustrup. *The C++ Programming Language*, Third Edition (Addison-Wesley, 1997).

Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library* (Addison-Wesley, 1999).

About the Author

Grum Ketema has Masters degrees in Electrical Engineering and Computer Science. With 17 years of experience in software development, he has been using C since 1985, C++ since 1988, and Java since 1997. He has worked at AT&T Bell Labs, TASC, Massachusetts Institute of Technology, SWIFT, BEA Systems, and Northrop.