DYNAMIC MEMORY ALLOCATION ON REAL-TIME LINUX

Jianping Shen
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
shen.jianping@foerstergroup.de

Michael Hamal
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
hamal.michael@foerstergroup.de

Sven Ganzenmüller
Institut Dr. Foerster GmbH und Co. KG
In Laisen 70, 72766, Reutlingen, Germany
ganzenmueller.sven@foerstergroup.de

Abstract

Dynamic memory allocation is used in many real-time systems. In such systems there are a lot of objects, which are referenced by different threads. Their number and lifetime is unpredictable, therefore they should be allocated and deallocated dynamically. Heap operations are in conflict with the main demand of real-time systems, that all operations in high priority threads must be deterministic. In this paper we provide a generic solution, a combination of the memory pool pattern with a shared pointer, which meets both: high system reliability by automatic memory deallocation and deterministic execution time by avoiding heap operations.

1 Introduction

Dynamic memory management on real-time multi-threaded systems has two handicaps.

1. Execution time for memory allocation and deallocation should be fast and predictable in time. Heap allocations in general (with `new()` or `malloc()`) are not deterministic, because of memory fragmentation and non-deterministic behaviour of system calls (`brk()`, `sbrk()`, `mmap()`) [1].
2. For objects which are referenced by more than one thread, it is difficult or even impossible to predict which thread will be the last user of the object and has the duty to deallocate the object's memory at the right time.

For such systems we need a deterministic, automatic and multi-threading capable dynamic memory management solution for C and C++ real-time development.

2 Detailed Problem Description

In this section we will explain the two mentioned handicaps in detail and try to figure out the solutions.

2.1 Execution Time of Memory Allocation

The implementation of memory management depends greatly upon operating system and architecture. Some operating systems supply an allocator for `malloc()`, while others supply functions to control certain regions of data. The same dynamic memory allocator is often used to implement both `malloc()` and `operator new()` in C++ [2].

Thus, we firstly limit our discussion to the following preconditions:

Operating system:	Linux
C Runtime Library:	glibc
Architecture:	X86-32

Linux uses virtual memory, each process runs in its own virtual address space. Linux dynamically maps the virtual memory to physical memory during runtime. The virtual memory layout for a running process is shown in figure 1.

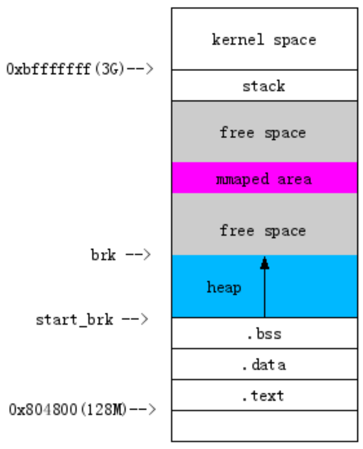

FIGURE 1: *Process memory layout on Linux.*

The allocator implementation in glibc is `ptmalloc2()`. A memory block managed by `ptmalloc2()` is called chunk. The heap organises all available chunks in two containers called *Bins* and *Fastbins*. Fastbins contains small sized[1] chunks for fast allocation, Bins contains normal sized[2] chunks. Available chunks in Bins are organized in 128 size-ordered double linked lists (shown in figure 2).

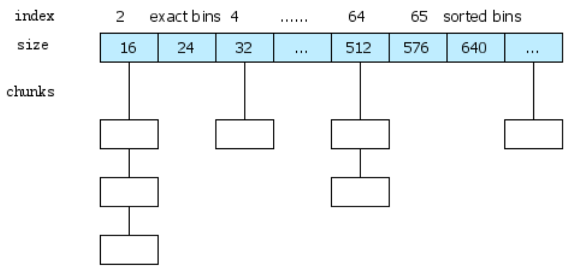

FIGURE 2: *Available chunks in Bin.*

If the deallocator function `void free(void* ptr)` is called, a chunk will be released. Which means the chunk is marked as available and dependent on its size, goes to Bins or Fastbins for further allocations. Obviously the available chunks in Bins and Fastbins are dynamic and dependent on runtime conditions.

If the allocator function `void* malloc(size_t size)` is called, the following steps will be processed:

1. If memory size $\leq$ `max_fast`, `ptmalloc2()` tries to find a chunk in Fastbins.
2. If step 1 failed or size > `max_fast` and size <= `DEFAULT_MMAP_THRESHOLD`, `ptmalloc2()` tries to find a chunk in Bins.
3. If step 2 failed, `ptmalloc2()` tries to increase the heap size by calling `sbrk()`.
4. If step 3 failed or size > `DEFAULT_MMAP_THRESHOLD`, `ptmalloc2()` calls `mmap()` to map a physical memory in the process virtual address space.
5. Finally return the allocated chunk, or `NULL` if steps 1 – 4 all failed.

This is a very simplified description. What really happens is much more complicated, but we don't want to inspect the details. It is now important to know, that execution time of memory allocation is not predictable.

For a memory allocation of small or normal size (steps 1 – 3), `ptmalloc2()` tries to find an appropriate chunk in Bins or Fastbins. The number of available chunks in both containers is dynamic and dependent on runtime conditions. The allocation time is therefore not predictable.

For large size memory allocations (step 4), `ptmalloc2()` uses the system call `mmap()` to map

[1]size <= `max_fast` (default 72 Bytes).
[2]size > `max_fast` (default 72 Bytes) and size <= `DEFAULT_MMAP_THRESHOLD` (default 128 kB).

physical memory in the process' virtual address space. With virtual memory management the physical memory may be swapped on hard disk. It involves disk IO, thus, its execution time is also unpredictable.

On other operating systems and architectures, the most allocator implementations involve system calls. Hence the memory allocation on such systems may be expensive and slow. The allocation time may or may not be predictable, depending on the concrete implementation.

A common solution for this problem is the memory pool approach. A memory pool preallocates a number of memory blocks during startup. While the system is running, threads request objects from the pool and return it back to the pool after usage. In a single threaded environment the memory pool pattern allows allocations with constant execution time [3]. In a multi-threading environment the execution time is predictable, but not constant.

2.2 Memory Deallocation at Right Time

Consider memory allocations in a multi threading environment. *Thread_A* allocates a memory block, and passes it as pointer to *Thread_B* and *Thread_C*. Now *Thread_A*, *Thread_B*, and *Thread_C* are all users of this memory block. The last user must deallocate the memory block to prevent a memory leak. Which of them becomes the last user is dynamic and depends on runtime conditions. In such a situation, its impossible to safely deallocate the memory by just calling the deallocator in one of the three threads.

A solution to this problem is the shared pointer pattern. The idea behind shared pointers (and other resource management objects) is called *Resource Acquisition Is Initialization* (RAII). Once a memory block is allocated, it will be immediately turned over to a shared pointer. A reference counter is used by the shared pointer to keep track of all memory block users. The shared pointer automatically releases the memory block when nobody is using this block any longer [4].

With the help of shared pointers, the users (in our case the threads) don't need to release the memory by themselves, this will be done by the shared pointer always at the right time.

3 The Approach

3.1 Combination of Memory Pool and Shared Pointer

The goal is to benefit from both approaches, a combination of the memory pool with shared pointers.

1. The memory pool preallocates a number of objects (see figure 3).
2. During runtime threads acquire objects from the pool as shared pointers.
3. If nobody is using the object any longer, the shared pointer will automatically return the object back to the pool (see figure 4).

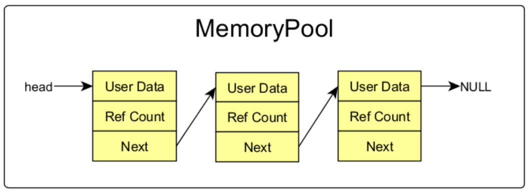

FIGURE 3: *Memory pool layout.*

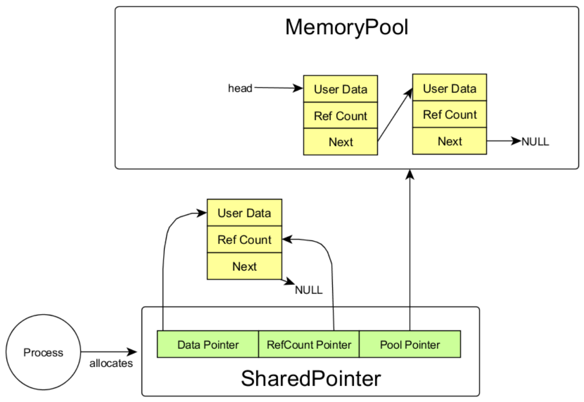

FIGURE 4: *Shared pointer layout.*

3.2 Execution Time of Shared Pointers

The common shared pointer implementation raises an interesting question: where's the reference counter located and who allocates it? Unfortunately, it will be allocated with `new()` on heap by the shared pointer itself. That means the constructor call of a shared pointer is unpredictable in time.

Hence we have to modify the shared pointer in a way that we also preallocate the reference counter as follows.

1. Extend the memory pool. The memory pool must preallocate memory not only for the object, but also for the object's reference counter.
2. Modify the standard shared pointer to work with the memory pool's preallocated reference counter.

3.3 The Final Approach

We put 3.1 and 3.2 together, and provide here the final solution.

1. The memory pool preallocates memory for user data and its reference counter.
2. At the runtime the process acquires memory from memory pool as shared pointer.
3. The Shared pointer uses the preallocated reference counter. Its execution time is therefore predictable. The Shared pointer keeps also a pointer to the memory pool. It will be used to return memory in step 4.
4. When nobody is using the memory block any longer, the shared pointer automatically returns the memory to memory pool.

This solution works without memory allocation at runtime. The process and the shared pointer both use preallocated memory from the memory pool. In our approach we call the shared pointer *RtSharedPtr*.

3.4 Execution Time

Our memory pool is designed to be used in multi-threading environments. The maximum execution time of a memory acquisition from a memory pool is dependent on the maximum number of threads. For multi-threading environments the access to the memory pool is protected by a mutex. Let's assume:

$$\begin{aligned} t_p &= \text{execution time for a memory} \\ &\quad\ \text{acquisition at a memory pool} \\ t_m &= \text{execution time} \\ &\quad\ \text{of mutex lock + unlock} \\ thread_{max} &= \text{maximum thread number} \\ t_{all} &= \text{complete latency} \\ &\quad\ \text{for a memory acquisition} \end{aligned}$$

Single-threaded environment:

$$t_{all} = t_p + t_m$$

The complete latency is constant in a single-threaded environment.

Multi-threaded environment:
Best case: no concurrent access to the memory pool

$$t_{all,min} = t_p + t_m$$

Worst case: full concurrent access, all threads access the memory pool at the same time.

$$t_{all,max} = (t_p + t_m) \cdot thread_{max}$$

Thus, the complete latency in a multi-threaded environment can be calculated as:

$$t_p + t_m \leq t_{all} \leq (t_p + t_m) \cdot thread_{max}$$

It is not constant, but predictable in time.

3.5 Performance Improvement

The mutex protection enforces that all parallel memory pool accesses will be serialised. This could be a performance bottleneck. A workaround is to create more memory pools to improve parallel memory acquisition.

If we create a memory pool for each dynamic data type in each thread, there is no concurrent memory pool access, thus, we don't need the mutex any more. We can disable the mutex protection by creating a memory pool as thread local. To create a thread local memory pool, we call the memory pool constructor as follows

```
RtMemoryPool(int size, bool tLocal=true,
const QString& name)
```

We will reach the maximal performance. The execution time is minimal and constant.

$$t_{all} = t_p$$

Obviously in a single-threaded environment we should create the memory pool as thread local to get the best performance.

4 The Implementation

4.1 Memory Pool

Our implementation[3] is based on C++ templates [5], therefore all preallocated memory chunks of a concrete memory pool are of the same size. That means, we need a memory pool for each dynamic allocated data type but the template approach garanties type-safety.

```
template <typename F>
class RtSharedPtr;

template <typename T>
class RtMemoryPool
{
    RtMemoryPool(int size, bool
        tLocal, const QString& name);
    RtSharedPtr<T> sharedPtrAlloc();
    ...
};
```

The preallocated memory blocks are organized as a linked list. Each block contains user data, its reference counter and a pointer to next block. For a memory acquisition the pool returns always the head block. (see figure 5)

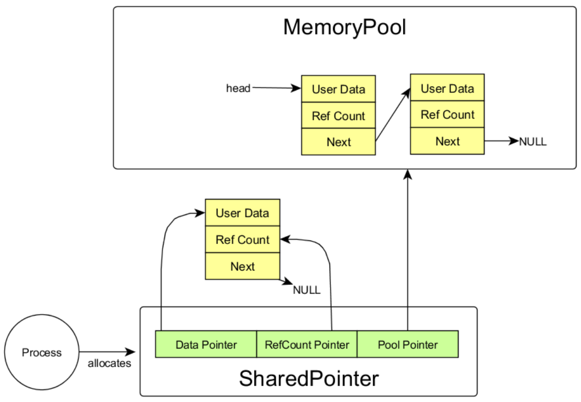

FIGURE 5: *Memory Acquisition*

If a memory block returns, it will be added as the new head block in front of the list. (see figure 6)

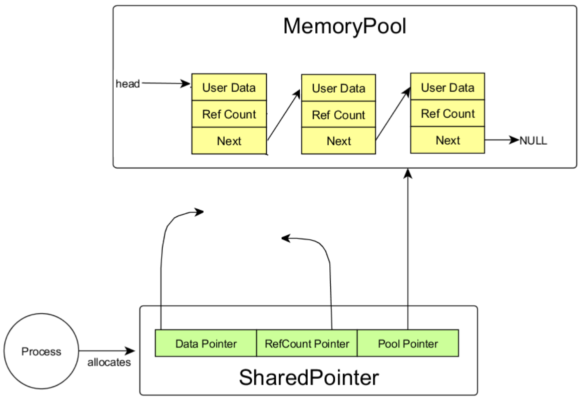

FIGURE 6: *Return Memory*

Instead of a raw pointer the memory pool returns a shared pointer to the allocated memory block, The shared pointer uses the reference count in the allocated memory block, and keeps a pointer to the memory pool.

```
//usage
RtMemoryPool<SomeType> memPool
    (100, "MyPool");
RtSharedPtr<SomeType> sptr =
    memPool.sharedPtrAlloc();
```

If the template type is a class, the memory pool will call the class constructor[4] to initialize the memory block. Accordingly the class destructor will be called when the memory block returns back to the pool. If the object holds some resources[5], they are released by the class destructor to prevent resource leaks.

4.2 Shared Pointer

We modified the boost shared pointer[6] as follows:

1. Make it possible to use the preallocated reference counter.
2. Add a boolean to indicate whether the managed memory block is from a memory pool.
3. Add a memory pool pointer, which will be used to return the memory block back to the pool.
4. Extend the release behavior; make the shared pointer also be able to return memory byck to the pool.

[3]Our implementation is based on the Qt library, so we use QString, which can be easily replaced by std::string.
[4]Constructors of pool objects should be kept simple, because their execution affects directly the memory aquisition time.
[5]Objects with dynamically allocated resources violate the real-time conditions because of their destruction in the object's destructor.

The modified shared pointer can work in two modes.

Normal mode: it works identically as a standard shared pointer and allocates the reference counter on heap. Its execution time is therefore unpredictable.

Real-time mode: in this mode the shared pointer must work together with a memory pool. Since the reference counter is preallocated in the pool, the execution time is predictable. There is no heap operation during runtime.

```
template <typename T>
class RtSharedPtr
{
    template <typename F>
    explicit RtSharedPtr(F* p) :
        pn(p), px(p)
    {
    }
    // a new constructor
    template <typename F>
    explicit RtSharedPtr(F* p,
            ReferenceCount* count,
            RtMemoryPool<T>* pool) :
        pn(p, count, pool), px(p)
    {
    } ...
}
```

We provide a new constructor. Compared to the boost version it takes two extra parameters. A pointer to a preallocated reference counter and a pointer to the related memory pool. With the new constructor the shared pointer will be bound to the memory pool and to it's preallocated reference counter.

A complete implementation can be found at http://www.foerstergroup.de/files/TS/RtSharedPtr.tgz

4.3 Test Results

We tested our solution on:

Hardware:	Intel Atom N270, 1 GB RAM
Operation System:	Linux 3.0.0-rt3
Compiler:	gcc/g++ 4.4.5
C-Library:	glibc 2.11.2

As test case we started two threads, one high-priority thread which allocates memory – in test case A from the memory pool (figure 7) and in test case B from heap (figure 8). The second thread acts as a noise generator, it allocates and frees memory chunks of different size on heap to bother the glibc's allocator.

The execution time for an allocation in the high-priority thread is meassured in μs and illustrated in figure 7 and figure 8.

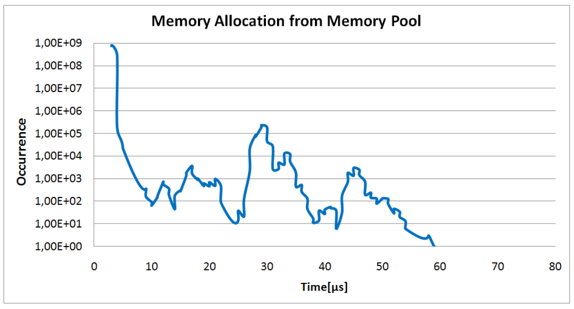

FIGURE 7: *Test case A: pool allocation.*

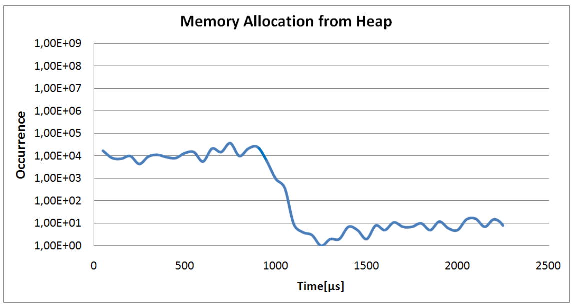

FIGURE 8: *Test case B: heap allocation.*

Both test cases run for approximately 10 hours and produce about 10^9 records.

Test case A shows, that 90% pool allocation is accomplished up to 5 μs, and 99.999% pool allocation up to 10 μs. The curve is limited to 60 μs on the time axis, which is the worst execution time. Our approach satisfies therefore hard real time systems timing requirements.

Compared to pool allocation, the execution time of a heap allocation is evenly distributed between 80 μs and 900 μs. Its worst excution time is not limited.

5 Conclusions

The described approach provides a generic solution, which meets both: high system reliability by automatic memory deallocation and deterministic execution time by avoiding heap operations.

We have tested our approach on Linux and Windows. As a generic solution it can be easily migrated to other operation systems, which do not provide operation system level memory allocation with deterministic execution time.

References

[1] Gianluca Insolvibile: *Advanced memory allocation*, 2003, LINUX JOURNAL ISSUE 109

[2] *http://en.wikipedia.org/wiki/Malloc*

[3] *http://en.wikipedia.org/wiki/Object_pool_pattern*

[4] Scott Meyers: *Effective C++, Third Edition*, 2005, PEARSON EDUCATION, INC.

[5] *C++ Deeply Embedded*, 2010, HILF GMBH.

[6] *http://www.boost.org/*