## Difficulty with Internal Fragmentation

We use a simple example to illustrate how this invention solves the memory internal fragmentation problem. First, let’s see a diagram of how existing memory allocation works today and understand the difficulty involved in solving the problem.



Figure 1 Current Memory Allocation

Figure 1 shows a typical example of an application using *malloc*()/*free*() calls to manage its memory, which consists of the following steps:

1. Call *malloc*() with desired memory size and get back a pointer *p* to the memory chunk
2. Use pointer *p* to access the content of the memory chunk
3. Call *free*() with pointer *p* to release the memory chunk when done

As indicated in the diagram, the memory chunk given to the application might happen to cross a page boundary. As a result, the application is occupying two pages. Even though the application might be just using a small portion at the top of the 2nd page, no other application can use the free space in the 2nd page. If we could move the memory chunk pointed by *p* “up” so that it falls completely inside the 1st page, then we could have recycled the 2nd page for use by other applications. Unfortunately, because the application is holding a direct pointer *p* to the memory chunk, there is nothing the memory allocator can do until the application calls *free*().

## Transparent Move of Memory Chunk

As the first step of addressing the challenge of moving an in-use memory chunk, this invention modifies the memory allocator such that a direct pointer *p* to the memory chunk like that in Figure 1 is not returned to the application. Instead, an indirect pointer that points to *p* is returned to the application. This is illustrated in Figure 2.

As indicated in the diagram, the pointer *p* returned to the application now does not directly point to the memory chunk. It points to the *realp* (the original *p* in Figure 1) that points to the memory chunk. Of course, in order to access the memory chunk, the correct code now must first dereference pointer *p* once to get *realp* before the first reference to *p*, as shown by the red line *p* = *p*->*realp* in Figure 2.



Figure 2 Indirect Memory Allocation

It would be impractical to require all applications to be changed for this additional line of code. This invention solves this problem by augmenting the compiler to track the pointer *p* returned by *malloc*() and to automatically insert the code *p* = *p*->*realp* without requiring any application change. With this indirection in place, the memory allocator can now move the memory chunk “up” into the 1st page and recycle the 2nd page, as shown in Figure 3. The move is transparent to the application since it’s pointer *p* never changes. The move is done by modifying *realp* to the new location of the memory chunk.



Figure 3 Transparent Move of Memory Chunk

## Synchronize Memory Chunk Access and Move

Another problem still remains when moving the in-use memory chunk. While the memory allocator is moving the chunk, the application could be actively accessing the chunk. So there is a race condition between the application and the memory allocator in terms of accessing *realp*. More specifically, the application needs to read the content of *realp* in order to access the chunk, while the memory allocator needs to modify the content of *realp* in order to move the chunk. And there can be multiple application threads accessing the chunk. So there must be concurrency control among the application threads and the memory allocator.

As the second step of addressing the challenge of moving an in-use memory chunk, this invention uses a simple read-write lock to solve the problem. The situation is a typical multiple readers single writer scenario. The application threads will only read the content of *realp*, while the memory allocator is the sole writer of *realp*. We extend the indirection mechanism in Figure 2 by associating with each *realp* a read-write lock *rwlock*, as shown in Figure 4.



Figure 4 Indirect Memory Allocation with Read-Write Lock (application side)

Before dereferencing pointer *p* to get *realp*, the application must acquire a read lock on *rwlock*. After the last reference to *p*, the application must release the read lock on *rwlock*. The additional code for performing the locking and releasing are indicated in Figure 4. Once again, these code are automatically generated by the compiler so no application change is required. This locking incurs little overhead to the application since all application threads are readers so they can proceed without blocking. The read lock on *rwlock* protects the application against the memory allocator from moving the chunk “underneath” it while it’s actively accessing the chunk.

On the memory allocator side, before it can move an in-use memory chunk, it must first acquire a write lock on *rwlock*. The write lock on *rwlock* protects the memory allocator against any application thread from accessing the chunk while it’s moving the chunk. The defragmentation logic of the memory allocator is shown in Figure 5.



Figure Indirect Memory Allocation with Read-Write Lock (allocator side)

## Summary

Our solution to the memory internal fragmentation problem is now complete. With the combination of returning indirect pointer to memory chunk and synchronizing among application threads and memory allocator using read-write lock, memory chunks can be freely moved around. Therefore, normal memory compaction using any existing algorithm can be performed by the memory allocator to reduce internal fragmentation. Using compiler support, additional application code required to dereference the indirect pointer and to acquire/release the read-write lock are automatically generated by the compiler, thus requiring no application change.