搜狗实验室技术交流文档

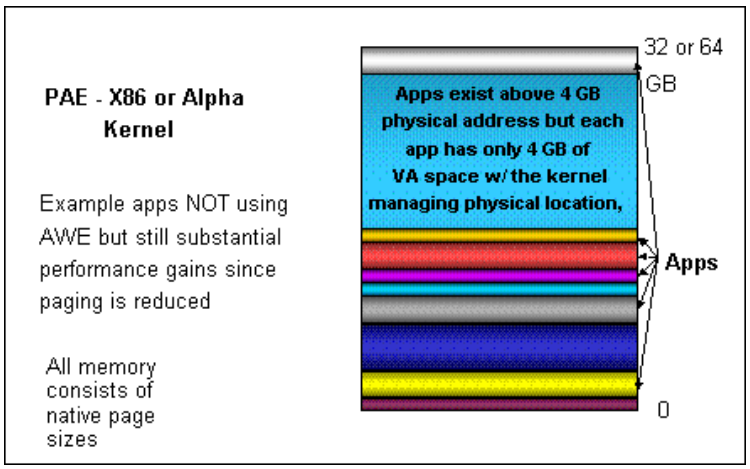
Vol4:1 Bigmem: 在32bit 下利用超过4G 内存

摘要

32bit 应用程序由于寻址空间的限制，无法直接使用 4G 以上的物理内存，这对一些性能要求高，内存开销大的应用程序而言是很大的限制. 本文介绍了一种在 32bit 下利用超过 4G 的内存的方法和相应的实现。

缘由

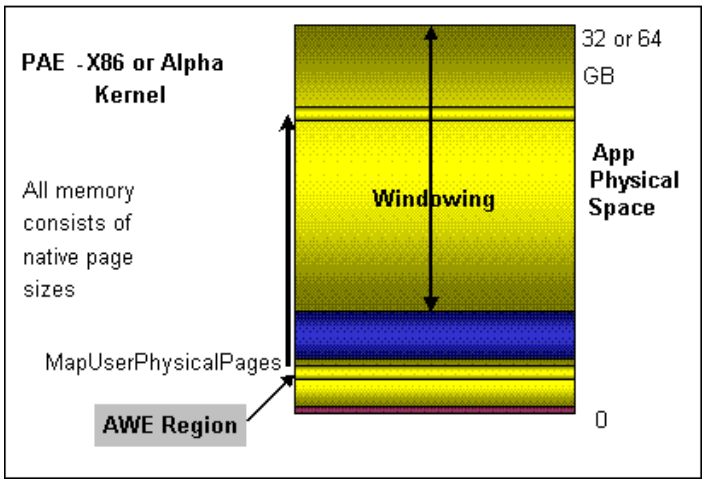
由于 PAE 技术的出现，现在的高端服务器往往配备有超过 4G 的物理内存。但是在单个 32bit 应用程序中最多只能访问不超过 3G 的线性地址空间。虽然 OS 可以利用剩下的内存作为文件系统缓存，但是当应用程序使用 O_DIRECT I/O 或者需要大片内存作为中间数据区，就会遇到不能充分使用系统内存的尴尬。



诚然，整体迁移到 64bit 系统是治标的方案。但是种种限制决定了我们很难轻易的作出这种选择——代码库太大很难转换、系统依赖的其他模块没有做好准备、需要集成的第三方软件(比如 JVM、MOD_PHP 等)没有做好准备、操作系统本身不够稳定、64bit 带来的程序执行效率降低、需要同时在 32bit/64bit 下部署统一的代码……。因此需要在 32bit 下有一套解决方案。

回忆历史，当年在 1M 寻址空间的 DOS 下，我们是如何利用 i386 主板上超过 1M 的物理内存？EMS 是一个经典的解决方案。EMS 在 1MB 空间中指定一块 64KB 空间作为窗口，分为 4 页，每页 16KB。EMS 存储器也按 16KB 分页，每次可交换 4 页内容，以此方式可访问全部 EMS 存储器。类似的，Windows 提供了 AWE(Address Windowing

Extensions) API，用于直接分配物理内存并将其映射到应用程序指定的窗口，从而使得应用程序可能使用到超过 4G 的物理内存。在 Linux 上，利用 file mapping 技术也可以达到类似的效果——在 ram disk (/dev/shm : tmpfs) 上建立文件或者通过辅助进程 lock memory 来保证内存页不被换出到 page file。我们把这套手段称为内存窗口技术。



内存窗口技术提供了突破 4G 限制的手段，但是仍然不够方便好用——应用程序需要灵活的内存分配手段，而窗口技术的要求严格按照页边界对齐，这带来很多麻烦。为了方便使用，在新版 local index 的开发过程中，我们实现了一套大内存的内存管理分配系统 bigmem api (暂时不向公司外人员公开代码)。bigmem api 本质上是一个在大文件中管理空间分配的方案。如果用 bigmem api 管理一个放在 tmpfs 中的文件，就达到了管理大内存的目的。bigmem 用 mmap 将当前需要访问的部分换入用户进程地址空间，由于 tmpfs 上的 mmap 不需要内存复制，这一过程的效率是较高的。

使用

提供了类似于 Win32 下 GlobalAlloc/GlobalLock API 的接口，在 bigmem.h 中可以查到 C 风格的 API，在 bigmem++.h 中可以找到功能更为丰富的 C++ API。

```
/**
 *    @param    bigmen_size:     the size of pool, in bytes
 *    @param    swap_filename:   the swap file path in system's tmpfs/shm
 */
HBIGMEMPOOL bigmem_init_create(uint64_t bigmem_size, const char* swap_filename, bool auto_del);
/**
 *    @param    bigmen_size:     the size of pool, in bytes
 *    @param    swap_fd:         the fd of swap file
 */
HBIGMEMPOOL bigmem_init_attach(uint64_t bigmem_size, int swap_fd);
/**
 *    close the pool
 */
int bigmem_fini(HBIGMEMPOOL pool);
```

```
/**
 *    check file
 */
int bigmem_check(HBIGMEMPOOL pool, bool showerror);
/**
 *    @param    n_bytes          The number of bytes to allocate
 *    @return                    a handle to memory
 */
HBIGMEM bigmem_malloc(HBIGMEMPOOL pool, size_t n_bytes);
/**
 *    @param    handle           The handle wanto free
 */
int bigmem_free(HBIGMEMPOOL pool, HBIGMEM handle);
/**
 *    Locks a local memory object and returns a pointer to the first byte of the object's memory block.
 */
void* bigmem_lock(HBIGMEMPOOL pool, HBIGMEM handle);
/**
 *    Decrements the lock count associated with a memory object
 */
int bigmem_unlock(HBIGMEMPOOL pool, HBIGMEM handle);
/**
 *    box a handle, use unbox_handle can get it again
 */
HBIGMEM_BOX bigmem_box_handle(HBIGMEMPOOL pool, BIGMEM_t* handle);
/**
 *    release a handle, but reserve the related object,
 *    if handle is boxed before, use unbox_handle to reobtain the handle.
 *    handle must not be locked
 */
int bigmem_release_handle(HBIGMEMPOOL pool, BIGMEM_t* handle);
/**
 *    unbox a handle, it must be boxed by box_handle
 */
BIGMEM_t* bigmem_unbox_handle(HBIGMEMPOOL pool, HBIGMEM_BOX& box);
```

bigmem_malloc 和 bigmem_free 函数对应于 libc 的 malloc 和 free，但是处理的是 HBIGMEM 而非 void*——如同 GlobalAlloc 返回 HGLOBAL 一样。用户得到 HBIGMEM 后不能直接使用，而是需要调用 bigmem_lock 将物理内存映射到进程空间，使用完毕后调用 bigmem_unlock 解除映射，这类似于 GlobalLock / GlobalUnlock。

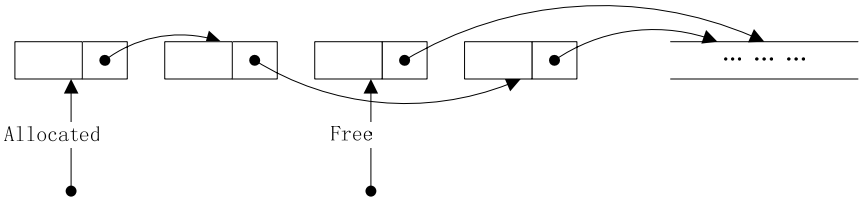
bigmem_init_create 按照指定的大小，创建一个受 bigmen 管理的交换文件。用 bigmem_init_attach 可以打开旧的 bigmem 文件，或者在进程间共享 bigmem。HBIGMEM 不能直接保存和传递，需要用 bigmem_box_handle 装箱，下次使用时用 bigmem_unbox_handle 重新得到 HBIGMEM。

出于性能考虑，bigmem api 本身没有任何锁来维护线程或者进程的同步，如果需要在多线程/多进程环境使用，用户必须自行维护一个锁。

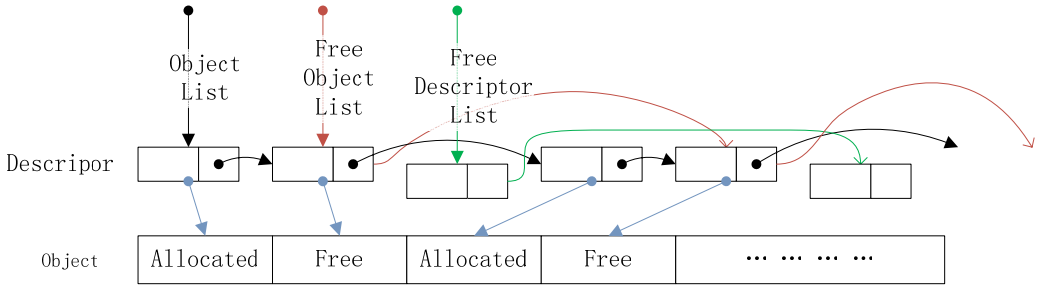
实现

文件数据结构

一般的内存分配器，是直接将被管理的内存块组织成链表，使用时遍历这个链表。



但是 bigmem 管理的是超过 4G 的大文件，如果也这样处理，会导致在分配和回收时需要访问散落在多处的数据（需要多次 mmap 或者 pread）。因此我们采取了将内存块和内存块的链表分开存储的方案。Bigmem 交换文件的 layout 如下：




```
# 类似 BNF 的文法
File ::= [File Header] ( [Descriptor Block][Object]* )+
File Header ::= [Head of Descriptor Block List][Head of Object List] \
      [Head of Free Object List][Head of Free Descriptor List]
Descriptor Block ::= [DBlock Header][Descriptor]{N /*a constant value*/}
DBlock Header ::= [Link of Descriptor Blocks List][Number of Descriptor]
Descriptor ::= [Link to Object][Object Size][Link of Object List] \
      ([Link of Free Object List] | [Link of Free Descriptor List])
Object ::= Bytes+
```

Object 是用来分配的内存块，有 Free 和 Alloced 两种状态。每个 Object 对应一个 Descriptor。Object 的管理信息由对应的 Descriptor 负责，Object 本身不存储任何管理信息。最开始的时候，文件除了 File Header 和 1 个 Descriptor Block 外，剩下的部分都属于唯一的一个 Free Object。

Descriptor 被集中存放在若干 128K 的 Descriptor Block 中，Descriptor 有 3 种状态：Free(不和任何 Object 关联)，WithFreeObject，WithAllocedObject。Descriptor 的状态由其所在的链表来区分：

	Object List	Free Object List	Free Descriptor List

Free			√
With Free Object	√	√	
With Alloced Object	√		

由于一个 Descriptor 不可能同时在 Free Object List 和 Free Descriptor List 中出现，所以在 Descriptor 中，这两个链表共享同一个“指针”。因为 Object Size 一定是 16 的倍数，所以 Descriptor 中 Object Size 最低位被借用来表示 Object 是否已经分配出去。

为了在调用 bigmem_free 时能够将被释放的 object 和其前后紧挨的 free object 合并，Object List 被要求严格按照 Object 在文件中的顺序排列。其他 2 个链表没有特殊要求。

HBIGMEM 句柄主要由 Object 偏移量、Descriptor 偏移量、映射到内存的地址组成。

以上具体的数据结构定义参见 bigmem_file.inl。链表的定义在 bigmem_list.inl，这部分内容学习自 linux kernel 的链表结构。

Code 实现细节

Bigmem 用 C++实现，类名为 BigMemPool，实现细节需要用到的数据在 BigMemPoolContext 中定义。

所有的 Descriptor Block 都被映射到内存，这样 malloc 和 free 操作都不需要再从外存 mmap，减少了开销。变量 page_map 存放文件偏移量到内存地址的映射，通过 off2descriptor 函数可以方便的得到一个 Descriptor 的内存地址。

malloc 时采用 best-fit 方法，和传统的内存管理器比起来，这里用了一些偷懒的技巧——用 stl set 做了一个 WithFreeObject Descriptor 的索引。索引以 Free Object 的 size 和 Descriptor 地址为 set 的元素。由于 stl set 是有序的，所以可以很快地得到和所需内存块大小最吻合的 Free Object。已知 Descriptor 的话，也很容易从 set 中反查。索引不会保存到文件中，而是在 init_attach 时 rebuild。

Malloc 内部提供了按页对齐的内存分配策略，这样 malloc 最多可能将一个 Object 分割为 3 个新的 Object：[不完整的一页 按页对齐被分走的一块 剩下的部分]。当 Fee Descriptor 少于 5 个时，malloc 会首先分配一个新的 Descriptor Block 以增加 Descriptor 数量。

Free 时，程序会从 Descriptor 的 Object List 链前后查看相邻的 Object，如果发现它们也 Free 的话，就会把相邻的 Free Object 合并为 1 个大的 Free Object。这一过程需要把待合并的 Descriptor 从索引中删掉。Free 后 Object 的 Descriptor 进入索引。

最后，check 函数用于检查 bigmem 是否一致。建议在 init_attach 后马上调用。

TODO

对小内存块采用 memory pool 来管理，避免较为昂贵的 descriptor 分配。

允许动态的增加文件大小。

集成一个小的 cache，避免调用 mmap 的次数过多。