Answer Set 4

19 APRIL, 1994

Introduction

The assignment is to design a memory management system for user programs. We are provided by the operating system with a contiguous block of memory, the *heap*, and must support user's requests for allocation and deallocation of memory from the heap. An allocation request will have a size parameter, and the memory management software must reserve a contiguous block of unused memory of this size from the heap. A pointer to the block is returned to the user. Deallocation calls are accompanied by a pointer to the block to deallocate. This memory is returned to the heap, meaning that it is available to be reused in a subsequent allocation request.

Statement of Design Goals

The most important goals are:

- Reliability.
- Speed.
- Economy of memory usage.

To economize on memory usage, we employ the trick of placing our free list data structures inside the free memory itself. We then try to minimize the header size of allocated blocks. In fact, we could set for ourselves the goal of having no header for allocated blocks, but we have decided not to do so for the following reason.

The type of an array in C does not include its final dimension. Hence a free cannot tell from the pointer type how much memory to free. It is possible to include this as part of the calling sequence, but the experienced applications' programmer would hide this housekeeping under a layer of data abstraction anyway. We therefore require a header for all allocated blocks giving the length of the block, and an additional "in-use" bit used for fragment reclamation.

We will not compactify the heap. Very few languages support compactification since it requires updating all pointers to items in the heap.

The requirement of speed gives use the most interesting challenge. We will use a singly-linked list of free blocks, since this has less overhead than a doubly linked list. Fragment reclamation will be done during forward sweeps of the free list only, hence reverse links will not be needed. It is too expensive to keep free blocks in-order on the free list, hence we will search for collatable fragments in memory. The compromise reached between speed and simplicity will be further discussed during the overview of the fragment reclamation algorithm.

Overview of approach

We assume that the operating system will provide us with the starting address and size of the heap, and will make provisions for the initialization of the memory management software. That is, either the user, the compiler or the operating system will call our initialization program `MemoryManagerInit` before running user code.

The heap will contain free and allocated blocks of memory. The free blocks are organized in a singly-linked list. Each block begins with a header. We consider the header of a free block to be inside the block, and the pointer to the free block's starting address is also the pointer to the header. The header of an allocated block will be before the block itself, and the pointer to the allocated block will point to the first byte after the header.

The last item on the free list will always be the rightmost free block, that is, the highest addresses in the heap. We will make the convention that this block is of infinite size, hence the free list is never empty. If the heap is a finite size, each allocate will guard against overflow by checking the range of address it wishes to return.

The allocated block header will contain the length of the block (not counting the header) and an *in-use bit.* The free block header will contain the the length of the block (including header), some flag bits: *in-use, swallowed, digesting,* and a pointer to the next free block header in the free list. The swallowed and digesting flags relate to the method of fragment reclamation.

We use first-fit to allocate memory. The free list is walked until the first block is found of size equal or larger than requested. If it is only slightly larger than requested, the entire block is allocated. There is a minimum value for the meaning of "slightly larger", since the resulting unused portion would not be large enough to hold a free block header.

To allocate the block we have three cases.

1. If the entire block is to be allocated, it is removed from the free list. The information from the free block header is reformated and written into the allocated block header, the pointer advanced to just after the header, and this pointer is returned to the user.

2. If the block is to be cut and allocated memory will be taken from the end of this block. In this case the only modification to the free list and free block header is that the length is updated. An allocated block header is created and the pointer following the header is returned to the user.

3. If the block is to be cut and allocated memory will be taken from the front of this block. The free block header is updated, moved rightwards, and the previous free block is updated. An allocated block header is filled out, placed at the front of this block and the pointer to the byte after the header is returned to the caller.

If the free block to cut is finite, cut from the end of the block. If it is infinite, cut from the front.

To free a block, we look just before the pointer to find the size of the allocated block being returned to the free list. The free block header is filled out, and we place this block at the front of the free list. We do not attempt to reclaim fragments at this time.

We will only attempt to reclaim, that is, glue together, fragments when an allocation call fails for lack of memory. We will reclaim and then try the allocate again. Reclamation will be performed in two forward sweeps over the linked list of free blocks. The first sweep will glue together free blocks, the second will delete from the free list blocks which have been "swallowed up" by reclamation.

The first sweep will visit each header in the free list, except the header of the last block on the list. If the swallowed bit is not set, the length will be added to the header pointer to arrive at the address of the byte following this free block. We check the in-use bit of this byte. If it is set, we are done. Else we have two side-by-side free blocks. The swallowed bit is set on the right free block (higher address), the digesting bit is set on the left free block (lower address), and the length of the digesting block is updated to the sum of the free block's lengths. We repeat this until either the block has infinite size or an allocated block sits to its right. We then continue with the next free block in the chain of free blocks.

The second sweep visits each header in the free list again. The free list item is deleted if the swallowed bit is set, and the digesting bit is always cleared. During the sweep, we are careful to maintain the invariant that the last item on the free list is the infinite sized free block containing the high addresses of the heap. This entry may try to "leap-frog" forward during the collation step.

We remark that the digesting bit does nothing for our algorithm but is included to ensure upward compatibility with an extension of the algorithm which may or may not be included, depending on the results of bench-marks. This extension is described in the next section. Again, it is important, if the digesting bit is not maintained, to at least minimize the coding impact if later we decide to implement this flag.

Extension to algorithm

A weak point of our algorithm is the batch nature of fragment reclamation. This lends to wide variations in call times. A method to spread out the work would employ the the digesting bit. Under this algorithm extension, freed blocks would immediately swallow free neighbors, and be marked digesting. Digesting blocks are in a transitional state: they are free but they cannot be allocated until it is sure that all the blocks it has swallowed have be removed from the free list. This can be assured if the free list contain no swallowed blocks. We keep a counter of the number of swallowed blocks on the free list. During the first-fit search of an allocation, remove from the free list any swallowed block found along the way, decrementing the counter. If the number of swallowed blocks becomes zero then you have two additional permissions:

1. To clear the digesting bit of any digesting block found on the free list during forward searches.
2. To allocate any appropriately sized digesting block.

If the swallowed block number is not zero, the algorithm cannot clear the digesting flag nor allocate digesting free blocks. If a full reclamation is called for, all the algorithm needs to do is sweep the free list deleting swallowed blocks and marking all other blocks as not digesting.

Calling Protocols

```
void MemoryManagementInit(void) ;
void * AllocMemory( long int size ) ;
/* Inputs: size of block to allocate
   Returns: a pointer to block else NULL if failed
      to appropriately sized memory
*/
void FreeMemory( void * p ) ;
/* Inputs: pointer to a allocated block. Must
      be a pointer previously returned by AllocMemory.
   Returns: none
*/
```

Data Structure Glossary

```
/* header for an allocated block */
typedef struct a_header {
    long int length ;
    /* in-use bit, MSB of length must = 0 */
    char first_user_byte ;
 } A_Header ;

/* header for a free block */
typedef struct f_header {
    long int length ;
    /* in-use bit, MSB of length must = 1 */
    struct {
      bit digesting ;
      bit swallowed ;
    } flags
    f_header * next ;
 } F_Header ;
```