Fall 2014 CS 33 Lab 4

DYNAMIC MEMORY ALLOCATION USING EXPLICIT FREE LISTS

In the lecture, we showed a dynamic allocator using implicit free lists with boundary tags. In this lab, we are going to implement an allocation simulator which uses an explicit free list with boundary tags. Review section 9.9.13 in the text to remember the details.

We will simulate malloc and free with our new functions: lalloc and lfree. Our simulation will use a heap which is an array of characters in our program. Our free list will be a linked list and our blocks will have boundary tags.

Here is the general structure:

The heap will have an initial size of HEAPSIZE of characters. And we will have a variable which maintains the current size of the heap: heapsize.

The heap will be made up of free blocks and allocated blocks. Each block will have a header and a footer. In between will be the “payload”: the area assigned after the call to lalloc.

We will have a data structure:

```
struct HDR          // free block header/footer/linked list
  {
  int  payload ;    // size of block (excluding headers)
  char freeall ;    // is this block allocated? 0=free/1=allocated
  int  succesr ;    // succeeding free block
  int previus ;    // previous free block
  } ;
```

which we will use as a template for both the header and footer. The succesr and previus fields will not be maintained for allocated blocks, and for the footer field of free blocks, although they will take up space in those blocks.

Our allocator will always return an area which is aligned on a 16 byte boundary and we will always allocate an area which is a multiple of 16 bytes. So, it is obvious that our header and footer will both take up 16 bytes, each, for the alignment and size roundoff reason. We are wasting some space but that will not be a factor in our implementation.

So, a block in our heap will look like:

HDR (16 bytes)
Payload (payload bytes)
Padding to 16
FTR (aka HDR) 16 bytes)

We will have a global variable: anchor, which is shaped like a header. It's payload will be zero, it is an allocated block and its succesr will point to the header of the first free block. Every free block will have a succesr pointing to the next free block. If the succesr pointer has a value equal to the current size of the heap, then it is the last free block.

Every block will have the payload in the header and footer and the freeall indicator set to 0 if it is a free block and 1 if it is an allocated block.

The **<u>init_heap</u>** will use calloc (we want the heap to be all zeroes) to allocate the heap, initialize the anchor and the whole block being the first free block. That is, with insert the header and footer.

The **<u>lalloc</u>** function will use first fit to find a block, rounding up the size to a multiple of 16 and to make space for the boundary tags. It will split the first fit block so that the lower address part of it is the assigned block and the upper part of it is the free fragment. If the block fits the request exactly, no splitting will be done. The header will save the payload for a subsequent lfree. Note that the actual size of each block is the payload plus 32 (the size of the header plus footer).

In lalloc, if there is no free block larger than the requested block (plus header/footer), it will use realloc to re-allocate the heap, adding HEAPSIZE additional space to the list as a free block. Zero out the payload area of the additional space. Repeat the process until there is enough free space to allocate the request. (Hint: you can use lfree to add the additional space).

The address returned by lalloc will be the integer location of the payload in the heap (not the header).

<u>lfree</u>: the obvious way to free an block is to set its “freeall” flag to 0 and add the block to the free block chain. The easiest way is to add it to the beginning of the chain: LIFO.

In lfree, we will use immediate coalescing. That is, if there is a free block on either side of the block being freed, it will be coalesced with that block. Recall that the footer of the previous block is directly adjacent to the header of the block being freed. Also, the header of the next block is adjacent to the footer of the block being freed.

Recall that there are 4 cases. Let A denote an allocated block and F, a free block, and the middle block is the one we are freeing:

```
            1 2 3
case 1: A A A        becomes A F A (no coalescing possible)
case 2: F A A        becomes F A where the F is 1 & 2 coalesced
case 3: A A F        becomes A F where the F is 2 & 3 coalesced
 case 4: F A F       becomes F  where the F is 1 & 2 & 3 coalesced and this is
                     just a combination of cases 2 & 3.
```

I have uploaded lab4.c as the framework.

Note that there are 4 macros:

```
hdr_payload(i)
hdr_freeall(i)
hdr_succesr(i)
hdr_previus(i)
```

They allow you to access a header or footer as if you are using a subscript inside heap. So, hdr_payload(16) accesses a header or footer starting at heap[16]. Notice that you must use parentheses instead of brackets because these are macros, not actual array accessing statements. If you are curious, use

```
gcc -save-temps lab4.c
```

and you can browse through the lab4.i file to see how this works.

I have given you the code for init_heap() so you will have an example of using headers and footers and the macros.

There is a dump_heap function which prints out the heap. It loops through the heap looking for headers and footers. It does this by checking the hdr_payload(i) for non-zero and printing it header/footer.

It would be advisable to zero out the payload of coalesced blocks (the ones which are absorbed) so that the dump_heap prints out only active headers and footers. It also prints a blank line in between every other line to make it easier to see headers and footers.

You will create the code for lalloc and lfree. The main() will run the following test cases:

1) allocate 20 blocks of random sizes.
2) runs lfree for cases 1-4 above.
3) attempts to allocate a block larger than the free space in the heap to exercise the re-allocate logic.
4) empties out the heap using by freeing the remaining allocated blocks.

Your submission will be the lab4.c and a text file which captures the output of the test run. The due date is December 10 at 6PM. But I encourage you to get this done earlier so as not to interfere with studying for the final.

YOUR PROGRAM MUST WORK ON SEAS LINUX!