Below is a descript of the implementation of mymalloc and myfree. After that is a description of the tests of memgrind to test our implementation and a discussion of results.

mymalloc.c

We use a static char array called memory of size 5000 to emulate a block of memory which malloc can allocate from.

The MAX allocation possible is 4998 bytes and the MIN allocation possible is 1 byte.

Stucture of blocks

memory array

0	blocksize	(allocated)
1	blocksize	
2	A	
3	A	
4	A	
5	-blocksize	(unallocated)
6	-blocksize	
7	U	
8	U	
9	blocksize	(allocated)
10	blocksize	
11	A	
.		
.		
.		
4990	A	
4991	A	
4992	A	
4993	A	
4994	blocksize	(allocated)
4995	blocksize	
4996	A	
4997	A	
4998	A	
4999	A	

mymalloc

blocksize includes the 2 bytes to store itself and the rest of the block blocksize is negative when space below is unallocated

To traverse the blocks you simply add the current index
to the current block's blocksize. The current index points
to the first of two bytes that store a blocks blocksize.

mymalloc traverses the blocks until it finds the first
unallocated block with a blocksize greater than or equal
to the requested size + 2 (for the 2 bytes needed for blocksize)
or requested size + 5 (see special case)

If the unallocated block is larger than the size needed, the
block is split in two, the first half being allocated, and the
second half unallocated.

myfree traverses the blocks until it finds the index where the
ptr given as an argument equals a pointer that points to the
array at that index.

Special case of mymalloc

The smallest blocksize allowed by our implementation is 3 bytes
because the smallest allocation allowed is 1 byte.
If a malloc call requests 10 bytes that allocation will search
for a block with blocksize 12. If it finds one then that block
is allocated. However, if the size is not an exact fit, then
a blocksize of 15 or higher is required because after splitting
the unallocated block, the latter block needs to be at least 3
bytes. If there ends up being unallocated blocks of 1 or 2 bytes
then they can never be allocated because they don't have enough room
to store a blocksize and contain data.

myfree

There are four possible cases.

Case 1: The block to be freed is between two allocated blocks

The size of the block to be freed is made negative to signify
that it is unallocated.

Case 2: The block to be freed is after an unallocated block

The the size of the block to be freed is added to the size of the previous unallocated block.

Case 3: The block to be freed is before an unallocated block

The size of the unallocated block after the block to be freed is added to the size of the block to be freed.

Case 4: The block to be freed is between two unallocated blocks

The size of the block to be freed and the size of the unallocated block after are added to the unallocated block before the block to be freed.

Case 5: The block is already unallocated

Accounting for these cases removes the possibility of two unallocated blocks from ever being next to each other

memgrind.c

This file contains the 4 tests required by the assignment instructions, two tests of our own design, and then a bonus test which is a copy of the code given in the assignment that shows what kind of situations should require our mymalloc and myfree functions to print an error message detailing invalid memory operations. Below is a short description of each function.

Each function completes the describes tests 100 times.

test1()

Calls malloc for 1 byte 3000 times. The pointers are stored in an array and then freed in the order that they were allocated.

test2()

Calls malloc for 1 byte and then frees the pointer immediately 3000 times.

test3()

Randomly choose between a malloc of 1 byte or a free(of a random pointer allocated). Once 3000 malloc calls have occurred the function stops calling malloc and frees any remaining pointers stored in an array randomly. If there are no pointers in the array when free is called then it tries to free a garbage value which myfree will catch with an error message.

test4()

Randomly choose between a malloc of random size from 1 to 4998 or a free(of a random pointer allocated). Once 3000 malloc calls have occurred the function stops calling malloc and freesany remaining pointers stored in an array randomly. If there are no pointers in the array when free is called then it tries to free a garbage valuewhich myfree will catch with an error message.

test5()

Calls malloc repeatedly of increasing size starting at 1 and incrementing by 1. Increments 50 times. Tests for memory saturation.

test6()

Calls malloc with pointers of various primitive type, assigns them a random value and then prints the values to show that the information is being stored in the array. Purpose of this test it to make sure values are not overwriting each other in memory.

bonustest()

These are the outputs of the the examples given in the assignment instructions.

Discussion

Test 1

Successful mallocs and total frees are both 166500 out of a total 300000 mallocs. This makes sense because the maximum amount of mallocs with size 1 is 1665 in an array of size 5000 with our implementation. This test takes the longest of the others but that is mostly due to the printing of error messages for the failed mallocs.

Test 2

All mallocs are successful in this test because there is never a chance that the memory array will be filled since every pointer is freed immediately.

Test 3

Although random, the chance of many more mallocs occurring vs frees is incredibly small, the memory never comes close to being freed because once a block is allocated it is freed not long after.

Test 4

Successful mallocs average around 180000-190000 out of 3000000. Since the average size of allocated is around 2500, there is enough space a little over half the time for allocation.

Test 5

This test fills the memory array with increasing malloc calls from size 1 up to 50. Nothing surprising here.

Test 6

This test shows that once a pointer is given a block of memory in the array that it can assign values to fill that memory and that those values are properly stored and there are no malformed memory locations.

Bonus Test

Should only print error messages corresponding to the detectable malloc and free calls outlined in the assignment instructions. Note that if you use ./memgrind | less the messages will not appear under the printf statements that says "Bonus test".