Computer Science 4500
Operating Systems
Fall 2018
Programming Assignment 3

Due on Thursday, November 29, 2018

Introduction

For this programming assignment you are to write a C or C++ program that demonstrates the processing of memory allocation and deallocation requests when the buddy algorithm is being used. The input data will begin with two integer values. The first of these indicates the amount of memory available for allocation. The second integer indicates the size of the smallest block that can be used to satisfy a request. Each of these integers will be a power of 2.

The remainder of the input will contain a sequence of allocation and deallocation requests, each on a line by itself. Each allocation request will have a unique integer ID which may be later specified in a request to deallocate the memory. Examples of allocation and deallocation requests are provided below, and sample input is provided on Loki. Your program is to demonstrate your understanding of the buddy system (and your programming prowess) by displaying the address associated with each requested memory allocation at the time it succeeds.

All allocations and deallocations in this assignment are simulated, and are to be assumed to take place in a region of memory beginning at address 0.

Basics

Recall that in the buddy system, each block used to satisfy an allocation request has a size that is exactly a power of 2 storage units (e.g. 512, 1024, 16384)[1]. The memory management software (i.e. your solution!) keeps lists of free blocks[2] (with the size of each free block equal to a power of 2) that are available for use in satisfying allocation requests.

Allocation of *K* bytes is performed by first rounding *K* up to the next power of 2 (if *K* wasn't already a power of 2), and checking the appropriate free list to determine if a free block of that size is available. If it is, then that free block removed from the list and used to satisfy the request, and the allocation is finished.

If the free list is empty, then the free list for blocks with the next larger size is examined. And if it doesn't have any entries, the system keeps looking in the free lists for larger and larger blocks, until either a free block is found, or the free lists for all the larger blocks have been examined.

---

[1] Although most modern machines are byte addressable, many earlier machines could only address larger units of storage, like 16, 24, 36, or 60 bits. We will assume bytes as the units of allocation to avoid further use of stilted terms like "units of storage."

[2] Obviously all the blocks on a single list will have the same size (which is some power of 2). Each list should also be kept in address order, from smallest to largest.

If no free block with the right size or a larger size is found, the allocation request isn't discarded. Instead, it is ***deferred*** until sometime later when a suitable block does become available[3]. Deferring a request is done by saving the request at the end of a list of such deferred requests. An example of a deferred request will be given later.

Of course, if a larger free block is found, then it is removed from its list and split into two smaller blocks (each half the size of the original), and these are placed on the appropriate free list. One of these blocks (we ***always*** choose the one with the smaller address[4]) is then used to satisfy the request, either directly, or indirectly, by splitting it again to yield two even smaller blocks which are handled in the same manner.

Deallocation is almost the inverse of allocation. When a block *B* is deallocated, its buddy's address is determined. Recall that if block *B* has address A and size S, we compute A / S, always yielding an integer result. If this value is even (that is, if the low-order bit in the binary representation of A / S is 0, or (A / S) mod 2 is 0), then the buddy's address is A + S; otherwise the buddy's address is A – S. The free list for blocks with size *B* is examined for a free block with the buddy's address. If that block is not found, then *B* is just added to that free list. Otherwise, *B* and its buddy (which is removed from the free list) are joined to form a larger free block *B′* (obviously twice as large as the original, with an address that is the smaller of the address of *B* and the address of its buddy). We then proceed exactly as if we were trying to deallocate this larger block *B′*. The process is guaranteed to terminate.

Once the deallocation is complete, we then need to determine if any of the deferred allocation requests can now be satisfied. This is done by considering each of the deferred requests, one at a time, in the same order the requests were submitted (that is, the deferred request queue is a FIFO queue). Each allocation request that can now be processed is removed from the deferred list (and appropriately allocated the needed storage).

[3] In a real system, the process making such a request might be blocked until sufficient storage becomes available to satisfy the request.

[4] This is done to guarantee that all correct solutions will yield the same output.

An Example

An illustration with real numbers will aid your understanding of these actions. Specifically, let's assume we have a region with 1024 bytes to use in satisfying allocation requests, and that the smallest region we'll use to satisfy a request is 128 bytes. (These numbers, 1024 and 128) would be the first two input values to the program.)

Then let's assume we have the following allocation and deallocation requests to process, in the order given.

ID	Action
1	Allocate 200 bytes
2	Allocate 399 bytes
3	Allocate 500 bytes
4	Allocate 100 bytes
5	Allocate 63 bytes
6	Allocate 75 bytes
4	Deallocate
5	Deallocate

Initially all our free lists are empty except for the one for size 1024 bytes, which has a single entry for the region starting at address 0.

The first request (ID 1, allocate 200 bytes, which is rounded up to 256 bytes) causes us to look first in the free list for 256 bytes (which is empty), then in the free list for 512 bytes (which is also empty), and finally the free list for 1024 bytes, which has one entry (for the free block at address 0). That entry is removed from the 1024-byte free list. Since it's larger than needed, it is split into two entries of size 512 (one at address 0 and one at address 512). The entry for address 512 is placed on the free list for size 512, and the one at address 0 is split again to give two free blocks of size 256, one at address 0 and one at address 256. The block at address 256 is added to the free list for size 256, and the one with address 0 is used to satisfy the current request.

The second request (ID 2, allocate 399 bytes, rounded up to 512) is easily satisfied, since there is a single block of size 512 available. After that request is satisfied, a single free block, of size 256, at address 256, remains.

The third request (ID 3, allocate 500 bytes, rounded up to 512) cannot be satisfied because there are no blocks on the free lists for 512 bytes or 1024 bytes. The request is therefore placed on the deferred list, and will be considered again after a deallocation.

The fourth request (ID 4, allocate 100 bytes, rounded up to 128) can be satisfied after the free block of 256 bytes at address 256 is split into two 128-byte blocks. The 128-byte block at address 256 is used to satisfy the request, and the other 128-byte block at address 384, is placed on the free list for 128 byte blocks.

The fifth request (ID 5, allocate 63 bytes, rounded up to 128, the minimum) is satisfied using the last free block in the system, that at address 384.

The sixth request (ID 6, allocate 75 bytes, rounded up to 128) cannot be satisfied immediately, so it is deferred, and placed after the first deferred request on the deferred list.

The seventh request (ID 4, deallocate) indicates that the 128 byte block at address 256 is to be freed. Since its buddy (at address 384) is not free, the free block is placed on the 128 byte free list. Since there are deferred requests, we consider each of them in order. The first such deferred request (for 512 bytes) still cannot be satisfied, so it remains on the list. The second deferred request (for 128 bytes) can be satisfied using the single free block at address 256. We now have no free blocks, and one deferred request remains.

The eighth and final request (ID 5, deallocate) frees the 128 byte block at address 384. Its buddy, at address 256, is not free, so we're done after placing the free block on the free list for 128 byte entries.

Note that the example given here and the additional example given later have all of the allocation requests first, followed by deallocation requests. This need not be the case in real test data. You are certainly encouraged to use these to test your program, but you should also prepare additional test cases. Developing suitable test data is an essential part of any development activity.

Details and Limits

As noted above, your program will begin by reading two positive integers, **`MSIZE`** and **`ASIZE`**, each of which will be a power of 2. **`MSIZE`** is the memory size that is to be managed by the buddy system, and also the size of the single free block (at location 0) that will exist when the simulation begins. **`ASIZE`** is less than or equal to **`MSIZE`**, and is the smallest block that is to be allocated to satisfy any allocation request. As a result, you will need one free list for each power of 2 between (and including) **`ASIZE`** and **`MSIZE`**, a list to keep track of deferred allocation requests, and some way of recording information (ID, address, and size) for allocations that have been made but have not yet been deallocated. The values of **`ASIZE`** and **`MSIZE`** will be such that they can be represented in a 32-bit unsigned integer.

After reading **`MSIZE`** and **`ASIZE`** and setting up the appropriate free lists, your program should read and process requests for allocation or deallocation, one at a time, until the end of input (end of file on standard input) is encountered. Each request will be on a single line that begins with a unique positive integer **`ID`** used to identify the request in the input and in the output. No two allocation requests will ever have the same **`ID`**, but each deallocation request must contain the **`ID`** of a previously successful allocation. The input data used to test your solution will never attempt to deallocate an allocation that is currently deferred, but your solution should detect this case if it should exist, since that would indicate either an error in the input data or, more troubling, an error in your solution! Although the **`ID`**s used in the earlier example were sequential, this need not be the case in actual test data.

After the **`ID`** in a request (and probably some whitespace – blanks or tab characters) there will be either a **`+`** or a **`-`** to indicate an allocation request or a deallocation request, respectively. In the allocation case, there will also be additional whitespace and the size of the region to be allocated. If the size is not already a power of 2, it should be rounded up to a power of 2, and it should also be at least as large as **`ASIZE`**. It should also never be larger than **`MSIZE`**, since such requests could never be satisfied.

After reading each request, your program should display a line that indicates the request that is being processed; they should look like one of these:

```
Request ID 52: allocate 73 bytes.
Request ID 35: deallocate.
```

Each line for an allocation request should be followed by

```
Success; addr = 0x00001000. total allocated size = 128
```

or

```
Request deferred.
```

as appropriate. Note that the address in the "Success" message should be the address at which the region allocated for the request begins. Also make certain that you always satisfy each allocation request with the free region that has the smallest address. (It will likely be appropriate to maintain the free lists in ascending address order to make this requirement easier to satisfy.)

After each deallocation request you should display a line that says

```
Success. total allocated size = 3072
```

Additionally, for each deferred request that can be successfully accommodated as a result of a deallocation, display a line that says

```
Deferred request with ID 59 allocated; addr = 0x00001800,
total allocated size = 4096
```

Of course, the IDs and address values in each of these is just an example. But do note that the addresses in the output should be displayed as 8-digit hexadecimal numbers preceded by 0x (that is, they should be C/C++-style hexadecimal constants).

Keep in mind that the addresses used in the program for free blocks and allocations are not real memory addresses – they are simulated memory addresses. ***Your program should not be doing dynamic memory allocation for blocks of the sizes used in the program.***

Naturally you *may* use dynamically-allocated data structures to represent the lists of free blocks of memory, the list of deferred allocation requests, and other data structures as needed by your solution.

To simplify your solutions, you may assume that there will never be more than 100 explicit allocation/deallocation requests in the input. That is, the input file will never contain more than 101 lines (the first line, of course, contains values for **`MSIZE`** and **`ASIZE`**). Additionally, you may also assume that there will never be more than 8 unique allocation sizes, the smallest of which will be **`ASIZE`**. So if **`ASIZE`** is (for example) 128, the 8 sizes you need to consider (and for which you need to maintain queues of available blocks) are 128, 256, 512, 1024, 2048, 4096, 8192, and 16384. This also means that **`ASIZE`** $\leq$ **`MSIZE`** $\leq$ **`ASIZE`** $\times 2^7$.

Sample Input and Output

The instructor's solution to this assignment and files containing sample input are available on Loki in the directory **`/home/phuang/csci4500/Fall2018/prog3`**. The instructor's solution is in the file named **`prog3`**. Like your solution should do, it reads input data from the standard input (file descriptor 0). You may also supply the option **`-v`** on the command line to produce considerable additional output. After each input request has been completed, the program will

display the status of all requests, the contents of the free lists, and the list of deferred requests. For deallocation, the calculation of buddy addresses and the joining of blocks on various free lists will be displayed.

The first sample is the input used the earlier discussion, and is found in the file **prog3.input1** on Loki. Here's what it looks like:

```
1024 128

1 + 200
2 + 399
3 + 500
4 + 100
5 + 63
6 + 75
4 -
5 -
```

The expected output for this input is as follows:

```
Request ID 1: allocate 200 bytes.
   Success; addr = 0x00000000, total allocated size = 256
Request ID 2: allocate 399 bytes.
   Success; addr = 0x00000200, total allocated size = 768
Request ID 3: allocate 500 bytes.
   Request deferred.
Request ID 4: allocate 100 bytes.
   Success; addr = 0x00000100, total allocated size = 896
Request ID 5: allocate 63 bytes.
   Success; addr = 0x00000180, total allocated size = 1024
Request ID 6: allocate 75 bytes.
   Request deferred.
Request ID 4: deallocate.
   Success. total allocated size = 896
   Deferred request 6 allocated; addr = 0x00000100, total
allocated size = 1024
Request ID 5: deallocate.
   Success. total allocated size = 896
```

The second sample is somewhat more ambitious, and is found in the file **prog3.input2** on Loki. Here's what it contains:

```
4096 256

1 + 500
2 + 250
3 + 100
4 + 1025
5 + 390
6 + 200
```

```
7 + 1
8 + 1200
9 + 200
10 + 2000
11 + 2000
4 -
3 -
2 -
1 -
9 -
7 -
5 -
6 -
8 -
10 -
11 -
```

And here is the expected output:

```
Request ID 1: allocate 500 bytes.
   Success; addr = 0x00000000, total allocated size = 512
Request ID 2: allocate 250 bytes.
   Success; addr = 0x00000200, total allocated size = 768
Request ID 3: allocate 100 bytes.
   Success; addr = 0x00000300, total allocated size = 1024
Request ID 4: allocate 1025 bytes.
   Success; addr = 0x00000800, total allocated size = 3072
Request ID 5: allocate 390 bytes.
   Success; addr = 0x00000400, total allocated size = 3584
Request ID 6: allocate 200 bytes.
   Success; addr = 0x00000600, total allocated size = 3840
Request ID 7: allocate 1 byte.
   Success; addr = 0x00000700, total allocated size = 4096
Request ID 8: allocate 1200 bytes.
   Request deferred.
Request ID 9: allocate 200 bytes.
   Request deferred.
Request ID 10: allocate 2000 bytes.
   Request deferred.
Request ID 11: allocate 2000 bytes.
   Request deferred.
Request ID 4: deallocate.
   Success. total allocated size = 2048
   Deferred request 8 allocated; addr = 0x00000800, total
allocated size = 4096
Request ID 3: deallocate.
   Success. total allocated size = 3840
```

```
   Deferred request 9 allocated; addr = 0x00000300, total
allocated size = 4096
Request ID 2: deallocate.
   Success. total allocated size = 3840
Request ID 1: deallocate.
   Success. total allocated size = 3328
Request ID 9: deallocate.
   Success. total allocated size = 3072
Request ID 7: deallocate.
   Success. total allocated size = 2816
Request ID 5: deallocate.
   Success. total allocated size = 2304
Request ID 6: deallocate.
   Success. total allocated size = 2048
   Deferred request 10 allocated; addr = 0x00000000, total
allocated size = 4096
Request ID 8: deallocate.
   Success. total allocated size = 2048
   Deferred request 11 allocated; addr = 0x00000800, total
allocated size = 4096
Request ID 10: deallocate.
   Success. total allocated size = 2048
Request ID 11: deallocate.
   Success. total allocated size = 0
```

Three additional test cases, appropriately named, are also provided. The expected output for each test case can be obtained using the instructor's solution. Additional test cases may be provided in the future. If so, they'll be announced on the class web page, and will be given names similar to the samples already provided.

Requirements

You must write (and test) a C or C++ program that performs the tasks associated with the buddy system memory management system to perform the requested specified in the input format described earlier. You should ideally have a single file of source code, appropriately named (that is, use **`prog3.c`** as part of the name). That file (or multiple files, if necessary) should be located in a top-level directory named **`csci4500-f18-prog3`** sometime before the due date at 11:59 PM. At that time, the contents of that directory will be copied to the instructor's directory for grading.

A "template" (partial solution) for a C solution to this assignment can be found in the **`csci4500`** directory. It is named **`prog3_temp.c`**. You are free to use this source code as the basis for your solution if you wish. You are not, however, required to use this partial solution. Locations in the code that are marked "**`XXX – WORK NEEDED`**" obviously will require additional work. There may also be other modifications required to the code.

You may work with one or two other students on this assignment. If you work in a group with others, you should clearly indicate the names of all individuals in the group in a comment near the top of the source code (you should indicate the author of your code in all cases, even if you are

working by yourself). You should also submit only one solution from the group for evaluation. Each individual in the group will receive the same evaluation.

As always, please contact the instructor if you have questions, and periodically check the class web site for any additions or corrections to this assignment.