CS 4375 Spring 2020 OS Lab 4 - Memory Allocator Evaluation

In this lab, you should evaluate three memory allocation strategies (first fit, next fit, best fit) by constructing a program that allocates and frees memory for an extended period of time. This program terminates and prints statistics on memory utilization and execution cost at the first time a memory allocation request fails.

Write the methods my_malloc(size) and my_free(address_pointer). The my_malloc() method takes as input the size of a memory allocation request. It takes a “heap” of memory and divides it into regions according to the size of the request and the allocation strategy. The heap memory is 1,048,576 bytes and does not change.

my_malloc(size) should return a pointer to memory of the desired size or NULL if the memory is not available. The memory allocated should start at a 16-byte boundary, i.e., the last digit of the memory address in hexadecimal should be zero. (See the section on alignment to powers of two below for hints.) my_malloc returns a pointer to the start of the allocated memory.

my_free(pointer) should free the memory pointed to by pointer if the memory was allocated by my_malloc(). If not, it should generate an error message and do nothing.

The allocation tests run in passes. For each pass, allocate 10 regions, numbered 0 to 9. The size of each region is taken from the rand.txt file provided. (The first entry in this file is 772. Your first allocation will be of size 772 bytes. The next entry in the table is 67. Your second allocation will be 67 bytes.) The values in this file are between 8 and 1008. At the end of each pass, all the odd numbered allocations are freed. Run until my_malloc() fails. Under optimal memory use, we would expect this failure to occur after about 420 passes.

When an allocation fails (and it will, since we only deallocate half of the allocated blocks), report the following:

The pass number
The size of the allocation that fails
The internal fragmentation (amount and fraction of memory lost)
The external fragmentation (amount and fraction of memory lost)
The amount of “overhead” memory used amount and fraction of memory used)
The total amount and fraction of memory allocated to processes (the part requested)

Review of overhead and fragmentation

- External Fragmentation: (free blocks)
  This is the available memory that are free, but cannot be allocated to user processes because the available "fragments" are smaller than the size requested.
- Internal Fragmentation
  This is the memory provided to user processes that they didn't request and therefore is wasted.
- Overhead:
  The amount and fraction of memory consumed by the arena's control blocks. Unlike fragments, this memory is **not** <u>wasted</u> since it is being used by the allocator. This is the total size of prefixes & suffixes.

Short note on Memory Alignment to Powers of Two

This document describes a very efficient technique used to round numbers represented in binary upwards to a multiple of a power of two.

Motivation

In order to simplify design and improve efficiency, some processors and i/o devices can only access data whose address is some multiple of a standard *alignment size*. For example, it is common for processors to only be able to load 8 byte (32 bit) integers from memory addresses that are a multiple of 8, and 16 byte (64 bit) integers from addresses that are multiples of 16. Alignment requirements for some i/o devices may be much larger, and are typically a power of two.

In order to facilitate management of memory in such a context, it is useful to be able to efficiently compute suitable addresses memory blocks.

Thinking about the problem in decimal

Tricks used in binary arithmetic (radix 2) often have analogues in decimal (radix 10), which is often more familiar.

Please consider the challenge of rounding a **decimal** integer i, say i=84, to the next multiple of ten (90). Call this desired operation align10(x). More formally, align10(x) computes the minimum multiple of 10 not less than x.

Before we solve this problem, consider that all multiples of $10=10^1$ have a zero as their least significant digit (as all multiples of $100=10^2$ have two zeros as their least significant digit) and it is particularly easy to *round down* to a lower power of ten (=80) by just clearing the least significant digit. Call this operation rounddown10(i). Clearly

- rounddown10(i) = i if i is a multiple of 10
- rounddown10(i) = i - i mod 10 otherwise

Why does this work: Because the *least significant **digit** of i* = i mod 10, and clearing the least significant digit is equivalent to substracting it out. Similarly, the *least two significant digits of i* = i mod 100, and for any power of ten 10^j the *j least significant digits of i* = i mod 10^j. Therefore, the same technique of clearing digits can be used to construct appropriate "rounddown" functions for any power of to

Now consider rounddown(x+9) for arbitrary x:

- rounddown10(x+9) = x if x is a multiple of 10
- rounddown10(x+9) = 10 + rounddown10(x) if x is not a multiple of 10
  Why: because adding 9 to x only causes a "carry" x is not a multiple of 10.

Now note that rounddown10(x+9) = align10(x).

Back to binary

Our objective is to define an efficient function align(a, x) which computes the minimum multiple of a not less than x where a is a power of two. Lets first consider some almost-familiar properties of binary numbers:

Clearly, since a is a power of two, then there exists a p such that a=2^p. Note that in binary, a can be represented as a string of 1's and 0's "10^p" meaning *1 followed by p zeros*, and all multiples of a have p zeros as lest-significant bits. For example 8=2^3=1000b, and 24=3*8=11000b. (henceforth I shall use a b to denote binary notation).

Now, consider a-1. Since a=2^p is a power of two, "a", when encoded in binary has the form of 1 followed by p zeros, and a-1 is encoded as p 1's. For example, if a=2^3=8=1000b, a-1=7=111b, and if a=$2^4$16=10000b, a-1=15=01111b, and if a=2^8=256=100000000b, a-1=255=11111111b.

So here's the first trick: let's define rounddown2(a,x) to be the largest multiple of a that is no greater than x. Clearly,

- rounddown(a, x) = x if x is a multiple of a
- rounddown(a, x) = x - x mod a

The problem with computing "mod" is that it requires division, which is expensive and slow to implement.

Once again, notice that x mod a where a=2^p is equal to the p least significant bits of x, and clearing these bits is equivalent to subtracting them out. Therefore, we just need to clear these low order bits, which can be done far cheaply & faster than using division.

Notice that a-1 has exactly the bits set that we want to clear in x. Recall that "~" is the "C" operator for *binary compliment*: **~(a-1)** has exactly the bits set that we don't want to change in x. Using this value as a *mask* and the *binary and* operator "&", we can clear the unwanted least significant bits, and select the bits we "want to save" by defining

```
        rounddown(a, x) = x & (a-1)
```

For example, consider the following six-bit representations of computations of rounddown(8, 24) and rounddown(8, 25):

```
        8 = 001000b
       24 = 011000b
        7 = 000111b
       ~7 = 111000b   (mask)
24 & (~7) = 011000b = 24

        8 = 001000b
       25 = 011001b
        7 = 000111b
       ~7 = 111000b   (mask)
25 & (~7) = 011000b = 24
```

The last step is to construct align(a, x) using rounddown. Recall that, like adding a "9" to a decimal quantity d only causes a decimal carry if d is not a multiple of 10. Similarly, if a = 2^p, then adding (a-1)+x will only generate a carry to the pth bit (or change any bit in position p or higher) if x is not a multiple of a. Therefore:

- rounddown(a, x+a-1) = x if x is a multiple of a
- rounddown(a, x+a-1) = x - x mod a + a otherwise]

Which is exactly the effect we wanted. Therefore, **iff a is a power of two**

```
        align(a, x) = (x + a - 1) & ~(a-1)
```

Working this through for a couple of examples:

```
align(8, 24):
        8 = 001000b
       24 = 011000b
   24 + 7 = 011111b = 31
       ~7 = 111000b
31 & (~7) = 011000b = 24

align(8, 25):
        8 = 001000b
       25 = 011001b
   25 + 7 = 100000b = 32
       ~7 = 111000b
32 & (~7) = 100000b = 32

align(8, 26):
        8 = 001000b
       26 = 011010b
  26 +  7 = 100001b = 33
       ~7 = 111000b
  33 & (~7) = 100000b = 32
```