Assignment 1

Zhizhang Deng (zd101), Xiaoxiao He (xh172)

CS214 – System Programming

Section 1: Methodology

1.1 Memory and Metadata Design

In this document, we will use a memory with 5 chunk (bytes) for demonstrating purposes.

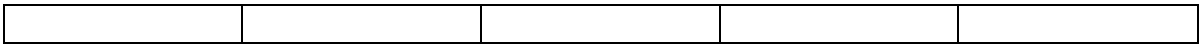

Fig. 1 Visualization of our memory model

We divide the 4096-byte memory into 4096 chunks, and our smallest unit for memory operation is a chunk (a byte) (Fig. 1). For each chunk, it is either a HEAD chunk, a SIZE chunk, or a DATA chunk (Fig. 2) (enforced by implementation).

Fig. 2 Demonstration of the inner structure of the memory

In Fig. 2, we call the metadata and the DATA chunk a memory block. This means a block contains the data and its metadata. HEAD and SIZE chunks are for storing necessary metadata, while the DATA chunk is for user to store information, here is our design for the HEAD chunk (and SIZE chunk).

HEAD Chunk:

There are two styles of HEAD chunk: HEAD (second HEAD in Fig. 2) and HEAD+SIZE (first HEAD in Fig.2). For each HEAD chunk, the format is the same: *abSSSSSS* (in bit).

'a' – The bit intended for determine whether the current memory block is being used. (0 – unused)

'b' – The bit intended for determine whether the current HEAD have a SIZE following it. (0 – no)

'SSSSSS' – Six bits for storing the size of the current memory block excluding the HEAD chunk in range $[0,64]$ bytes.

When the size of the current memory block excluding the HEAD chunk is larger or equal than 65 bytes, we need a SIZE chunk for storing the additional bits. Therefore, the HEAD and SIZE structure is like this: *abSSSSSS* | *SSSSSSXX*. (X stands for unused space)

Our implementation of metadata is significantly better than other design in memory consumption by using variable length of the size. We reduce one chunk of memory for each block with data size less than 64 bytes, which is better than fixed length (taking 2 chunk of memory).

1.2 Functioning Logic

In this section, we will discuss the functioning design of our program. First, every chunk of the memory either is a DATA chunk or is a HEAD (or SIZE) chunk, which means there are no memory chunk that doesn't have a metadata. This step seems to be unnecessary, but by using this unnecessary step, we enforced the operation to close mathematically, which means either you free or you malloc, we always operate on a HEAD chunk.

When we free a block, we will consolidate (if available) the next freed memory to ensure maximize memory use.

SOME DATA	HEAD|SIZE	HEAD|SIZE	SOME DATA
SOME DATA	FREED HEAD | SIZE	HEAD|SIZE	SOME DATA
SOME DATA	FREED HEAD | SIZE | 63		SOME DATA

For testing, we use clock instead of the sophistic gettimeofday to increase the precision and reduce the difficulty of implementation.

In this malloc assignment, we divide our project into several sub procedure:

1. We utilize the struct architecture for reading regularized metadata information.
   For Example:

```
PACK(
        struct header {
                unsigned char is_used : 1;
                unsigned char is_large : 1;
        }
);
```

   We use the PACK keyword for eliminating padding among the variables since we need to read consequential bits.
2. We modular our functioning logic to the following sub procedure:
   a. Adjust_header_size: Intended for calculating the header size for concatenating and allocating memory.
   b. Write_new_header: Intended for writing new header with given parameters into the memory.
   c. Get_previous_header: Intended for get previous header, return NULL when not found.
   d. Get_next_header_position: Intended for get next header position, return NULL when not found.

Section 2: Testing for Robustness

Here is one testing case:

SOME DATA	HEAD|31	HEAD|32	SOME DATA
SOME DATA	FREED HEAD | 31	HEAD|32	SOME DATA
SOME DATA	FREED HEAD | SIZE | 63		SOME DATA

This is a special case since 63 is apparently not a size that we need to request an additional SIZE chunk, but if we allocate |HEAD|64|, we need additional one SIZE chunk, however, the limited free memory space cannot provide additional space. Therefore, we could only assign the memory in this fashion |HEAD|SIZE|63| so that nothing bad happened.

Workload E:

We randomly malloc block of size 1-100 until the memory is full, then we write random things into the block and free the block. We repeat this procedure for 500 times, and our malloc passed the test.

Workload F:

We first randomly malloc block of size 1-100 until the memory is full, then we write random things into the block. After that we iterate the following instruction until the memory has been cleared:

1. We randomly free 50 blocks (if possible)
2. We randomly malloc 25 blocks with size ranging from 1-100.
3. Write things to 25 blocks.
4. Go to 1

This will eventually free all memory and is intended to simulate all possible working scenario. This test plan is different from E because in this test plan, we will free memory between two allocated blocks, and will test the concatenating nearby unused memory space, which is essential for maximizing memory usage.