CS105 – Computer Systems Spring 2023

Week 11: Dynamic Memory

April 10-12, 2023

1. Consider an allocator that uses an implicit free list. Assume that all blocks (both allocated and free) contain a 32-bit header and a 32-bit footer. Assume that each block has a total size (including header and footer) that is a multiple of 8 bytes, thus only the 29 higher order bits in the header and footer are needed to record block size. The usage of the remaining 3 lower order bits is as follows:

   - `bit 0` indicates the use of the current block: 1 for allocated, 0 for free.
   - `bit 1` indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
   - `bit 2` is unused and is always set to be 0.

   Given the contents of the heap shown on the left, show the new contents of the heap (in the right table) after a call to `free(0x400b010)` is executed. Your answers should be given as hex values. Note that the address grows from bottom up. Assume that the allocator uses immediate coalescing, that is, adjacent free blocks are merged immediately each time a block is freed.

Address	
0x400b028	0x00000012
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	0x00000012
0x400b018	0x00000013
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	0x00000013
0x400b008	0x00000013
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	0x00000013

Address	
0x400b028	
0x400b024	0x400b611c
0x400b020	0x400b512c
0x400b01c	
0x400b018	
0x400b014	0x400b511c
0x400b010	0x400b601c
0x400b00c	
0x400b008	
0x400b004	0x400b601c
0x400b000	0x400b511c
0x400affc	

2. Assume you are running on the same system as in the previous problem. Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines the code section label that implement the corresponding functionality correctly.

```
/* given a pointer p to an allocated block, i.e., p is a pointer returned by some
previous malloc() call; returns the pointer to the header of the block */
void * header(void * p){
    void *ptr;
    _______;
    return ptr;
}

A. ptr=p-1
B. ptr=(void *)((int *)p-1)
C. ptr=(void *)((int *)p-4)
```

```
/* given a pointer to a valid block header, returns the size of the block */
int size(void *hp){
    int result;
    _______;
    return result;
}

A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)
```

```
/* given a pointer p to an allocated block,i.e. p is a pointer returned by some
previous malloc() call; returns the pointer to the footer of the block */
void * footer(void * p){
    void *ptr;
    _______;
    return ptr;
}

A. ptr=(void *) ((char *)p+size(header(p))-8)
B. ptr=(void *) ((char *)p+size(header(p))-4)
C. ptr=(void *) ((int *) p+size(header(p))-2)
```

```
/* given a pointer to a valid block header, returns the usage of the currect block,
1 for allocated, 0 for free */
int allocated(void * hp){
    int result;
    ______;
    return result;
}

A. result=(*(int *)hp)&1
B. result=(*(int *)hp)&0
C. result=(*(int *)hp)|1
```

```
/* given a pointer to a valid block header, returns the pointer to the header of
previous block in memory */
void * prev(void *hp){
    char *ptr;
    ______;
    return (void *) ptr;
}

A. ptr = (char *)hp - size(hp)
B. ptr = (char *)hp - size(hp-4)
C. ptr = (char *)hp - size(hp-4) + 4
```