CSAPP-Malloc Lab Report

1. 评分

2. 总体说明

   采用分离显式链表，首次适配策略，32 位程序，双字对齐。分离链表设置了 20 个类别，16 字节及以下为第一类，之后按照 2 的次方分类。

3. 数据结构

   3.1 堆的宏观结构

Head of Class1	Head of Class2	…	Head of Classn	Padding (Optional)	Prologue 8/1	Prologue 8/1	Block	…	Block	Epilogue 0/1

Block 代表分配和空闲块，其他块均为四字节大小。分离列表中有 n 个类别，每个类别的链表头指针依次排列在最前面，而 heap_listp 始终指向 Head of Class1。之后是可能存在的对齐 Padding，取决于 n 的奇偶性，紧跟着的是序言块(8 字节已分配)，最后是结尾块（0 字节已分配）。

   3.2 块的微观结构

Free blocks:

31 3 2 1 0

Header: Block size | a/f
Predecessor
Successor
Free
Padding(optional)
Footer: Block size | a/f

Allocated blocks:

31 3 2 1 0

Header: Block size | a/f
Payload

Padding(optional)	
Footer: Block size	| a/f

其中，Free 和 Payload 不代表具体大小，其他均为 4 字节。

4. 主函数设计

4.1 mm_init

初始化，申请空间，同时构建上述堆结构中除了 Block 以外的部分。

首先申请空间用于这些部分：

```
/* Create the initial empty heap */
if ((heap_listp = mem_sbrk((4 + CLASSNUM) * WSIZE)) == (void *) -1)
    return -1;
```

初始化所有分离链表头地址为 0：

```
for (int i = 0; i < CLASSNUM; i++)
{
    PUT(heap_listp + i * WSIZE, 0);
}
```

字节对齐填充块，初始化序言块以及结尾块：

```
PUT(heap_listp + CLASSNUM * WSIZE, 0);                  /* Alignment
padding */
PUT(heap_listp + ((1 + CLASSNUM) * WSIZE), PACK(DSIZE, 1)); /*
Prologue header */
PUT(heap_listp + ((2 + CLASSNUM) * WSIZE), PACK(DSIZE, 1)); /*
Prologue footer */
PUT(heap_listp + ((3 + CLASSNUM) * WSIZE), PACK(0, 1));     /*
Epilogue header */
```

准备好这些后，进行第一次用于 payload 的空间申请：

```
/* Extend the empty heap with a free block of CHUNKSIZE bytes */
if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
    return -1;
return 0;
```

4.2 mm_malloc

根据对齐要求等，将用户输入的字节数转换为内部的实际大小，同时利用 find_fit 和 place 找到合适的空闲块进行放置。若没有，申请额外空间。

首先做一些基本检查：

```
if (heap_listp == 0)
{
    mm_init();
}
/* Ignore spurious requests */
if (size == 0)
    return NULL;
```

然后，根据字节对齐调整大小：

```
/* Adjust block size to include overhead and alignment reqs. */
if (size <= DSIZE)
    asize = 2 * DSIZE;
else
    asize = DSIZE * ((size + (DSIZE) + (DSIZE - 1)) / DSIZE);
```

准备齐全，尝试寻找合适的空闲块，如果找到，进行放置：

```
/* Search the free list for a fit */
if ((bp = find_fit(asize)) != NULL)
{
    place(bp, asize);
    return bp;
}
```

如果没找到，我们需要请求更多空间，请求的大小是我们当前需要的空间，和之前我们设定的一次请求至少多大空间的较大者，同时进行放置。

```
/* No fit found. Get more memory and place the block */
extendsize = MAX(asize, CHUNKSIZE);
if ((bp = extend_heap(extendsize / WSIZE)) == NULL)
    return NULL;
place(bp, asize);
return bp;
```

4.3 mm_free

本质上直接对接合并空闲块。

首先处理异常情况：

```
if (bp == 0)
    return;
if (heap_listp == 0)
{
    mm_init();
}
```

调整 bp 的 header 和 footer 为 free：

```
size_t size = GET_SIZE(HDRP(bp));
PUT(HDRP(bp), PACK(size, 0));
PUT(FTRP(bp), PACK(size, 0));
```

进入合并完成剩下的操作：

```
coalesce(bp);
```

4.4 mm_realloc

没有特殊的设计，处理特殊情况加上之前的 malloc 和 free

首先处理新大小为 0 的情况，直接 free 掉：

```
size_t oldsize;
void *newptr;

/* If size == 0 then this is just free, and we return NULL. */
```

```
if (size == 0)
{
    mm_free(ptr);
    return 0;
}
```

处理原指针为 NULL 的情况：

```
/* If oldptr is NULL, then this is just malloc. */
if (ptr == NULL)
{
    return mm_malloc(size);
}
```

根据请求大小，分配一个新指针并做异常处理

```
newptr = mm_malloc(size);

/* If realloc() fails the original block is left untouched  */
if (!newptr)
{
    return 0;
}
```

复制旧数据，我们需要先判断新和旧的哪一个更小，然后用 memcpy 复制这一部分内容：

```
/* Copy the old data. */
oldsize = GET_SIZE(HDRP(ptr));
if (size < oldsize) oldsize = size;
memcpy(newptr, ptr, oldsize);
```

最后释放掉原指针即可：

```
/* Free the old block. */
mm_free(ptr);

return newptr;
```

5. 辅助函数

5.1 extend_heap

根据字节对齐要求，增加堆大小，释放原结尾块并设置新的，尝试合并空闲块。

首先根据请求的字数，计算并调整出合适的字节数，并进行请求：

```
char *bp;
size_t size;
/* Allocate an even number of words to maintain alignment */
size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
if ((long) (bp = mem_sbrk(size)) == -1)
    return NULL;
```

改造原本的结尾快为我们新请求出来的一整个大空闲块的 header，补上 footer，同时构建新的结尾块。

```
/* Initialize free block header/footer and the epilogue header */
PUT(HDRP(bp), PACK(size, 0));         /* Free block header */
PUT(FTRP(bp), PACK(size, 0));         /* Free block footer */
PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* New epilogue header */
```

进行空闲块合并和入链：

```
/* Coalesce if the previous block was free */
return coalesce(bp);
```

5.2 locate

找到适合目前请求大小的分离链表的类别。

按照一开始说明的类划分方法进行计算即可：

```
int i = 4;
for (; i <= 22; i++)
{
    if (size <= (1 << i))
        return i - 4;
}
return i - 4;
```

5.3 push

将一个空闲块加入合适的大小类的空闲链表，并且永远直接加入空闲链表的头。分两种情况处理：空链表和非空链表。

首先确定应该在哪一类空闲链表：

```
size_t size = GET_SIZE(HDRP(bp));
int index = locate(size);
```

如果这一类链表是空的，那么 bp 就应当作为链表头，同时 bp 没有前节点，也没有后节点。

```
/* Case 1: Empty list */
if (GET_LIST(index) == NULL)
{
    PUT(heap_listp + WSIZE * index, (unsigned int) bp); /* bp as head
*/
    PUT(bp, 0);                                  /* No predecessor
for bp */
    PUT((unsigned int *) bp + 1, 0);                /* No successor
for bp */
}
```

如果已经存在，我们的策略是直接插入在链表头部，那么原本的头节点现在是 bp 的后节点，同时 bp 也是原头结点的前节点。bp 没有前节点，把存放的头节点改为 bp。

```
/* Case 2: Already existent list, add to the head */
else
{
    PUT((unsigned int *) bp + 1, (unsigned int) GET_LIST(index)); /*
```

```
Original head as bp's successor */
    PUT(GET_LIST(index), (unsigned int) bp);                    /* bp
as the original head's predecessor */
    PUT(bp, 0);                                                 /* No
predecessor for bp */
    PUT(heap_listp + WSIZE * index, (unsigned int) bp);         /* bp
as new head */
}
```

5.4 delete

将块从空闲链表删除，需要分为四种情况，依次考虑：链表中唯一的块，在链表尾端，在链表头部，在链表中间。

首先还是确定类别：

```
size_t size = GET_SIZE(HDRP(bp));
int index = locate(size);
```

最特别的情况是，这个类别的链表只有 bp，那么把头节点归零即可。

```
/* Case 1: Only bp in the list */
if (GET_PRED(bp) == NULL && GET_SUCC(bp) == NULL)
{
    PUT(heap_listp + WSIZE * index, 0); /* Set head to null */
}
```

如果 bp 是最后一个节点（链表不止一个节点），把 bp 的前节点的后节点设为 0，就能让 bp 脱链

```
/* Case 2: bp as the last one */
else if (GET_PRED(bp) != NULL && GET_SUCC(bp) == NULL)
{
    PUT(GET_PRED(bp) + 1, 0); /* Set the successor of bp's
predecessor to null */
}
```

若 bp 是头节点（链表不止一个节点），那么我们需要把新的头节点设置成 bp 的后节点，同时把 bp 的后节点的前节点设为 0，让 bp 脱链

```
/* Case 3: bp as the first one */
else if (GET_SUCC(bp) != NULL && GET_PRED(bp) == NULL)
{
    PUT(heap_listp + WSIZE * index, (unsigned int) GET_SUCC(bp)); /*
Change head to bp's successor */
    PUT(GET_SUCC(bp), 0); /* Set bp's successor to null */
}
```

最一般的在中间的情况，需要修改 bp 的前节点的后节点和 bp 的后节点的前节点

```
/* Case 4: bp in the middle */
else if (GET_SUCC(bp) != NULL && GET_PRED(bp) != NULL)
{
    PUT(GET_PRED(bp) + 1, (unsigned int) GET_SUCC(bp)); /* Set bp's
predecessor's successor to bp's successor */
```

```
    PUT(GET_SUCC(bp), (unsigned int) GET_PRED(bp)); /* Set bp's
successor's predecessor to bp's predecessor */
}
```

5.5 place

放置分配块，同时尝试对占用的空闲块进行分割。

放置时，将空闲块从链表移除：

```
size_t csize = GET_SIZE(HDRP(bp));
delete(bp);
```

如果空闲块比此时请求的大出了最小块或更多，我们对空闲块进行分割：

```
if ((csize - asize) >= (2 * DSIZE))
{
    PUT(HDRP(bp), PACK(asize, 1));
    PUT(FTRP(bp), PACK(asize, 1));
    bp = NEXT_BLKP(bp);
    PUT(HDRP(bp), PACK(csize - asize, 0));
    PUT(FTRP(bp), PACK(csize - asize, 0));
    push(bp);
}
```

把前一部分的大小设置为我们此时请求的（已分配），后一部分设为剩余大小（未分配），让 bp 指向后一部分同时加入空闲链表。

如果多处的空间并不足以放下最小快，则不进行分割：

```
else
{
    PUT(HDRP(bp), PACK(csize, 1));
    PUT(FTRP(bp), PACK(csize, 1));
}
```

5.6 find_fit

采用 first-fit 策略，寻找合适的空闲块。

首先寻找一个最小的满足要求的类：

```
int index = locate(asize);
unsigned int *bp;
```

然后，先对这个类的链表进行遍历，采用 first-fit 策略，找到第一个大小满足要求的就采用。如果在这个类别里没找到，则继续去下一个更大的类别中找，重复这个过程直至找完最大的类别。

```
while (index < CLASSNUM)
{
    bp = GET_LIST(index);
    /* First-fit */
    while (bp)
    {
        if (GET_SIZE(HDRP(bp)) >= asize)
        {
```

```
            return (void *) bp;
        }
        bp = GET_SUCC(bp);
    }
    /* Check a bigger class */
    index++;
}
```

如果还没找到，返回 NULL，malloc 函数会申请空间。

```
return NULL; /* No fit */
```

5.7 coalesce

合并空闲块，对于一个空闲块 bp，分下列情况：如果前后都是已分配块，则只是把 bp 加入空闲链表；如果前后有空闲块，或者前后都是空闲块，那么把合并后的新空闲块加入链表，同时删除原有的。

首先明确前后块是否已分配：

```
size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
size_t size = GET_SIZE(HDRP(bp));
```

根据我们先前 free 的设计，如果前后都是已分配块，我们需要把 bp 加入空闲块链表：

```
/* Case 1 */
if (prev_alloc && next_alloc)
{
    push(bp);
    return bp;
}
```

如果前分配后空闲，将后块移除空闲链表，其大小加入 bp 中

```
/* Case 2 */
else if (prev_alloc && !next_alloc)
{
    delete(NEXT_BLKP(bp));
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
}
```

如果前空闲后分配，将后块移除空闲链表，其大小加入 bp 中，同时把 bp 调整为前块的位置，这里现在是整个新空闲块的首端。

```
/* Case 3 */
else if (!prev_alloc && next_alloc)
{
    delete(PREV_BLKP(bp));
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
```

```
    bp = PREV_BLKP(bp);
}
```

对于前后都空闲，结合以上两种情况，删除这两个空闲块，大小都并入 bp，并让 bp 指向新的空闲块首端。

```
/* Case 4 */
else
{
    delete(NEXT_BLKP(bp));
    delete(PREV_BLKP(bp));
    size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
           GET_SIZE(FTRP(NEXT_BLKP(bp)));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
    bp = PREV_BLKP(bp);
}
```

对于 2，3，4，我们需要把新的 bp，也就是新的大空闲块首端加入空闲链表：

```
push(bp);
return bp;
```

6. 检查

非常基础的检查，显示位于堆最前面的链表头信息

```
static void printlist(void)
{
    for (int i = 0; i < CLASSNUM; ++i)
    {
        printf("%p:%p\n", heap_listp + WSIZE * i, GET_LIST(i));
    }
}
```

检查块的合法性

```
static void checkblock(void *bp)
{
    if ((size_t) bp % 8)
        printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
        printf("Error: header does not match footer\n");
}
```

显示块的信息：

```
static void printblock(void *bp)
{
    size_t hsize, halloc, fsize, falloc;

    checkheap(0);
    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
```

```
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));

    if (hsize == 0)
    {
        printf("%p: EOL\n", bp);
        return;
    }

    printf("%p: header: [%zd:%c] footer: [%zd:%c]\n", bp,
          hsize, (halloc ? 'a' : 'f'),
          fsize, (falloc ? 'a' : 'f'));
}
```

显示堆中的块的信息，检查序言和结尾块。

```
static void checkheap(int verbose)
{
    char *bp;

    if (verbose)
        printf("Heap (%p):\n", heap_listp + CLASSNUM * WSIZE);

    if ((GET_SIZE(HDRP(heap_listp + CLASSNUM * WSIZE)) != DSIZE)
|| !GET_ALLOC(HDRP(heap_listp + CLASSNUM * WSIZE)))
        printf("Bad prologue header\n");
    checkblock(heap_listp + CLASSNUM * WSIZE);

    for (bp = heap_listp + CLASSNUM * WSIZE; GET_SIZE(HDRP(bp)) > 0;
bp = NEXT_BLKP(bp))
    {
        if (verbose)
            printblock(bp);
        checkblock(bp);
    }

    if (verbose)
        printblock(bp);
    if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp))))
        printf("Bad epilogue header\n");
}
```