第九章 虚拟内存 家庭作业

——1173000825 李大鑫

9.11

虚拟地址 0x027c

A:虚拟内存地址格式

|--------------TLBT----------------|---TLBI----|

0	0	0	0	1	0	0	1	1	1	1	1	0	0

B:地址翻译

参数	值
VPN	0x09
TLB 索引	0x01
TLB 标记	0x02
TLB 命中？（是/否）	否
缺页？（是/否）	否
PPN	0x17

C:物理地址格式

0	1	0	1	1	1	1	1	1	1	0	0

|----------------CT-----------------|------------CI-----------|-----CO----|

D:物理地址引用

参数	值
字节偏移	0x00
索引缓存	0x0F
缓存标记	0x17
缓存命中？（是/否）	否
返回的缓存字节	-

9.13

虚拟地址：0x0040

A:虚拟内存地址格式

|--------------TLBT----------------|---TLBI----|

0	0	0	0	0	0	0	1	0	0	0	0	0	0

B:地址翻译

参数	值
VPN	0x01
TLB 索引	0x01
TLB 标记	0x00
TLB 命中？（是/否）	否
缺页？（是/否）	是
PPN	-

C:物理地址格式

访问页面未分配，触发错误 Page Fault

D:物理地址引用

访问页面未分配，触发错误 Page Fault

9.15

块大小：前部需要

请求	块大小（十进制字节）	块头部（十六机制）
malloc(3)	8	0x9
malloc(11)	16	0x11
malloc(20)	24	0x19
malloc(21)	28	0x21

9.17

基本思路和代码与书上的实现方式相同，修改首次适配搜索为下一次适配搜索。如何实现下一次适配：只需要使用一个全局变量 rover，用来记录上一次“下一次适配搜索”搜索的终点 block，然后这次搜索从 rover 这一 block 开始即可。

本着不重复造轮子的原则，这里使用 malloc lab 中提供的测试策略对代码进行测试。

代码如下:

```
/*
 *  @author: lidaxin
 *  @since : 12/11/2018
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include "mm.h"
#include "memlib.h"

/*
 * If NEXT_FIT defined use next fit search, else use first fit search
 */
#define NEXT_FIT x     //设置NEXT_FIT 进行下一次适搜索

/* Team structure */
team_t team = {
#ifdef NEXT_FIT
       "implicit next fit",
#else
       "implicit first fit",
#endif
       "Dave OHallaron", "droh",
       "", ""
};

/* $begin mallocmacros */
/* Basic constants and macros */
#define WSIZE      4      /* word size (bytes) */
#define DSIZE      8      /* doubleword size (bytes) */
#define CHUNKSIZE  (1<<12)  /* initial heap size (bytes) */
#define OVERHEAD    8      /* overhead of header and footer (bytes) */

#define MAX(x, y) ((x) > (y)? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p)      (*(size_t *)(p))
```

```
#define PUT(p, val)  (*(size_t *)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)      ((char *)(bp) - WSIZE)
#define FTRP(bp)      ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) -
WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) -
DSIZE)))
/* $end mallocmacros */

/* Global variables */
static char *heap_listp;  /* pointer to first block */
#ifdef NEXT_FIT
static char *rover;      /* next fit rover */
#endif

/* function prototypes for internal helper routines */
static void *extend_heap(size_t words);
static void place(void *bp, size_t asize);
static void *find_fit(size_t asize);
static void *coalesce(void *bp);
static void printblock(void *bp);
static void checkblock(void *bp);

/*
 * mm_init - Initialize the memory manager
 */
/* $begin mminit */
int mm_init(void)
{
    /* create the initial empty heap */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
        return -1;
    PUT(heap_listp, 0);                          /* alignment padding */
    PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
    PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
    PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));   /* epilogue header */
```

```
    heap_listp += DSIZE;

#ifdef NEXT_FIT
    rover = heap_listp;
#endif

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

/*
 * mm_malloc - Allocate a block with at least size bytes of payload
 */
/* $begin mmmalloc */
void *mm_malloc(size_t size)
{
    size_t asize;     /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size <= 0)
        return NULL;

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

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

    /* 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;
}
/* $end mmmalloc */

/*
 * mm_free - Free a block
 */
/* $begin mmfree */
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}

/* $end mmfree */

/*
 * mm_realloc - naive implementation of mm_realloc
 */
void *mm_realloc(void *ptr, size_t size)
{
    void *newp;
    size_t copySize;

    if ((newp = mm_malloc(size)) == NULL) {
        printf("ERROR: mm_malloc failed in mm_realloc\n");
        exit(1);
    }
    copySize = GET_SIZE(HDRP(ptr));
    if (size < copySize)
        copySize = size;
    memcpy(newp, ptr, copySize);
    mm_free(ptr);
    return newp;
}

/*
 * mm_checkheap - Check the heap for consistency
 */
void mm_checkheap(int verbose)
{
```

```
    char *bp = heap_listp;

    if (verbose)
        printf("Heap (%p):\n", heap_listp);

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

    for (bp = heap_listp; 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");
}

/* The remaining routines are internal helper routines */

/*
 * extend_heap - Extend heap with free block and return its block
pointer
 */
/* $begin mmextendheap */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;

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

    /* 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 */
```

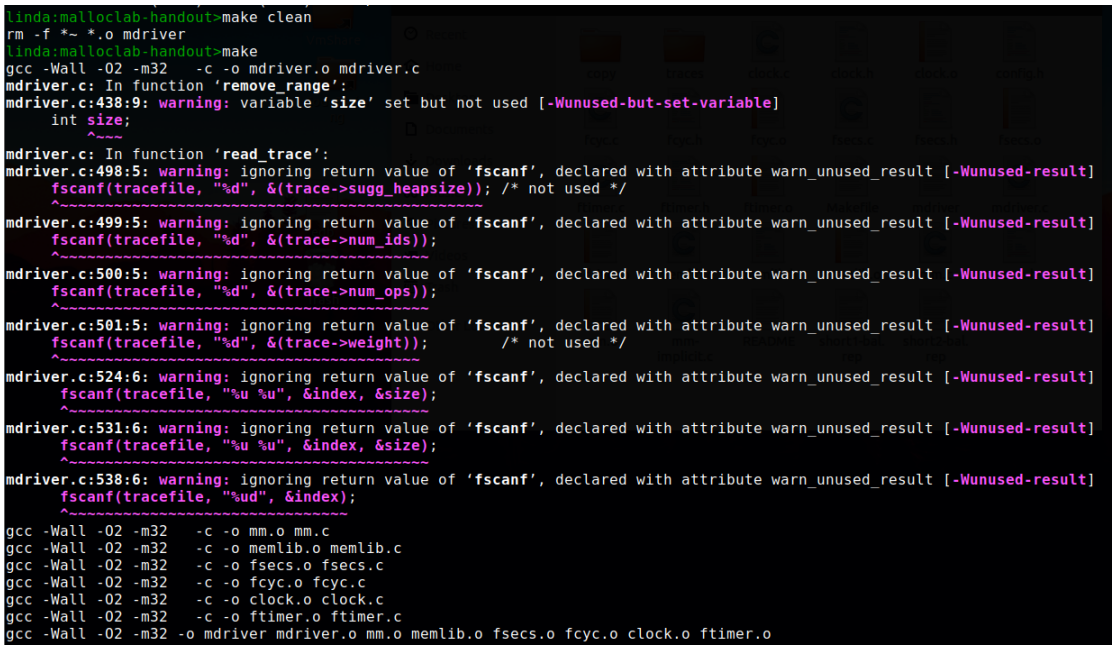
```
    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
/* $end mmextendheap */

/*
 * place - Place block of asize bytes at start of free block bp
 *         and split if remainder would be at least minimum block size
 */
/* $begin mmplace */
/* $begin mmplace-proto */
static void place(void *bp, size_t asize)
/* $end mmplace-proto */
{
    size_t csize = GET_SIZE(HDRP(bp));

    if ((csize - asize) >= (DSIZE + OVERHEAD)) {
        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));
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
/* $end mmplace */

/*
 * find_fit - Find a fit for a block with asize bytes
 */
static void *find_fit(size_t asize)    //寻找可以放置目标数据的block [放置策略]
{
#ifdef NEXT_FIT
    /* 放置策略:  下一次适配搜索 */
    char *oldrover = rover;

    /* rover记录上次搜索的终点 从rover开始搜索到end */
    for ( ; GET_SIZE(HDRP(rover)) > 0; rover = NEXT_BLKP(rover))
    if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
```

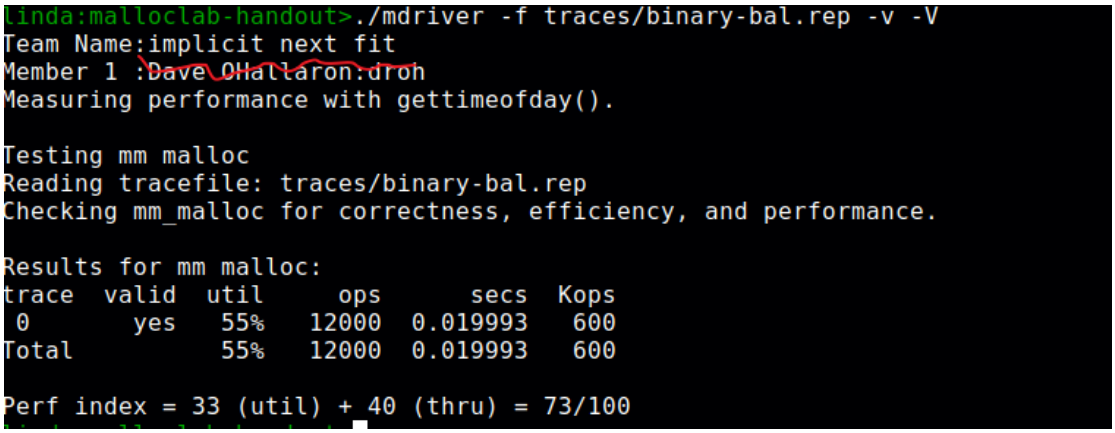
```
        return rover;

    /* 如果没有找到 从开头搜索到rover 循环*/
    for (rover = heap_listp; rover < oldrover; rover =
NEXT_BLKP(rover))
    if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
        return rover;

    return NULL;  /*无法放置*/
#else
    /* 放置策略: 首次适配搜索*/
    void *bp;
    //从头开始搜索能够放置的地方
    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp))
{
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
            return bp;
        }
    }
    return NULL;   /*无法放置*/
#endif
}

/*
 * coalesce - boundary tag coalescing. Return ptr to coalesced block
 */
static void *coalesce(void *bp)
{
    /* 合并空闲块 */
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp))); //获得前面的
block
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp))); //获得后面的
block
    size_t size = GET_SIZE(HDRP(bp));

    if(prev_alloc && next_alloc) {      //case1: 前后都不空闲
        return bp;
    }
    else if(prev_alloc && !next_alloc) {    //case2: 前面不空闲 后面空闲
对后方进行合并
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));  //改变size
        PUT(HDRP(bp),PACK(size,0));           //修改HDRP block
        PUT(FTRP(bp),PACK(size,0));           //此时bp的FTRP已经是原来后一
块的FTRP 修改FTRP block
```

```
    }
    else if(!prev_alloc && next_alloc) {    //case3: 前面空闲，后面不空闲
对前方进行合并
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));  //改变size
        PUT(FTRP(bp),PACK(size,0));            //修改FTPR
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));  //修改前方block 的 HDRP
        bp = PREV_BLKP(bp);
    }
    else {                                  //case4 : 前后都是空闲块 都需要进行合
并
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
GET_SIZE(FTRP(NEXT_BLKP(bp)));  //改变size
        PUT(HDRP(PREV_BLKP(bp)),PACK(size,0));  //改变前方HDRP  此过程中
bp是不会改变的
        PUT(FTRP(NEXT_BLKP(bp)),PACK(size,0));  //改变后方FTRP
        bp = PREV_BLKP(bp);
    }
    return bp;
}


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

    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: [%d:%c] footer: [%d:%c]\n", bp,
          hsize, (halloc ? 'a' : 'f'),
          fsize, (falloc ? 'a' : 'f'));
}

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");
}
```

代码测试：





9.19

1）选 **a**

a：正确。对于一个 2^(m+1)大小的块，申请存放一个(2^k)+1 大小的空间，此时伙伴系统的所有空间都被占用，而实际上其中一个“伙伴”只存放了大小为 1 的数据。

b：错误。最佳适配算法是要扫描所有的块，然后选择一个“浪费最少”的块，而首次匹配算法从开始的块扫描，只要扫描到的块比要放入的数据大就放置，所以相比而言，首次适配

算法比最佳适配算法要快一些（平均）。

c：错误。P603 提出了使用 LIFO 的顺序维护链表，将新释放的块放置在链表的开始处。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块，在这种情况下，释放一个块可以在常数的时间内完成。所以不只有地址递增。使用边界标记的方法来回收才会快速。

d：错误。不一定。因为伙伴系统进行划分的时候是按照地址切半划分的，所以当两个相同大小的块地址并不连续的时候，代表这两个块其实并不是“伙伴”，所以不能进行合并。这就会产生虽然有足够空间，但是不能存放要求数据的外部碎步。

2）选 **d**

a:错误。因为这种情况下首次分配的算法会很快找到匹配。这种方法的确有利于解决外部碎片的问题，因为通过分割、与后面的块合并操作 会始终保证在开始的是最大的块，这样会解决因为空闲快无法合并而导致的外部碎片问题。

b:错误。应该按照块大小从小到大排会比较好。

c:错误。最佳匹配要选的是最终空闲空间最小的块。

d:正确。如果块大小递增，则根据两个算法的原理，“空闲空间最小的块”和“第一个能够放下目标数据的块”应该是同一个块，所以两者等价。

3）选 **b**

根据书上的原话，引用如下:

C 程序的 Mark & Sweep 收集器必须是保守的，其根本原因是 C 语言不会用类型信息来标记内存位置。因此，像 int 或者 float 这样的标量可以伪装成指针。例如，假设某个可达的已分配块在它的有效载荷中包含一个 int，其值碰巧对应于某个其他已分配块 b 的有效载荷中的一个地址。对收集器而言，是没有办法推断出这个数据实际上是 int 而不是指针。因此，分配器必须保守地将块 b 标记为可达，尽管事实上它可能是不可达的。