Lab5实验报告

本次实验为五次实验最难的一次实验，本实验报告仅按照书上的版本进行了实现(find\_fit和place函数是我自己实现的)，不过我会将提供书上代码的十分完整的注释版，且进行了一些小小的优化，可使分数提高。

无论是隐式空闲链表还是显式空闲链表（或者分离式空闲链表），根据malloclabPDF的要求，任何复合的数据结构都是不允许的（数组，树，结构体等），只能使用普通的数据类型，因此在书上，mm.c中使用了static char\*heap\_listp作为隐式空闲链表的头节点，建立在此基础上，开始以下代码讲解：

首先是书上的宏：

#define WSIZE 4 //书上自定义的字、脚部或头部的大小（字节），intel中2个字节是一个字，4个字节是双字

#define DSIZE 8 //书上自定义的双字大小（字节）

#define CHUNKSIZE (1<<12) //扩展堆时的默认大小（4096/WSIZE）

#define MINBLOCK (DSIZE + 2\*WSIZE)//设置最小块的大小（块内容（有效荷载+填充）+头部（一个字）+尾部（一个字）），尽管整个书上的代码并没有用到

#define MAX(x, y) ((x) > (y) ? (x) : (y))//比较x和y的大小，返回较大的那个，相等的话返回y

#define PACK(size, alloc) ((size) | (alloc)) //将 size 和有效位合并为一个字

#define GET(p) (\*(unsigned int \*)(p)) //读地址p处的一个字

#define PUT(p, val) (\*(unsigned int \*)(p) = (val)) //写地址p处写一个字，写为val

#define GET\_SIZE(p) (GET(p) & ~0x07) //得到地址p处的 size（GET得到该处的字，接着与上~0x07，即0xfffffff8，将size取出）

#define GET\_ALLOC(p) (GET(p) & 0x1) //得到地址p处的有效位（是否分配，取最低位）

//bp=block point(指向有效荷载)

#define HDRP(bp) ((char\*)(bp) - WSIZE) //获得头部的地址

#define FTRP(bp) ((char\*)(bp) + GET\_SIZE(HDRP(bp)) - DSIZE) //获得脚部的地址, 利用GET\_SIZE取得块大小，加上这个size后减去头部和脚部的大小(总共为双字)

#define NEXT\_BLKP(bp) ((char\*)(bp) + GET\_SIZE(((char\*)(bp) - WSIZE))) //计算后块的地址，利用头部较近的特点，算出size，此时直接加上这个size就行了，因为中间跨过了所在块的脚部和下一个块的头部

#define PREV\_BLKP(bp) ((char\*)(bp) - GET\_SIZE(((char\*)(bp) - DSIZE))) //计算前块的地址

我们跳过头文件：

#define ALIGNMENT 8//这条宏将该分配器设置为8字对齐

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)//将size伸展成对齐的大小

#define SIZE\_T\_SIZE (ALIGN(sizeof(size\_t)))//将size\_t的大小调成和对齐标准一样

static char\* heap\_listp;//指向序言块

mm\_init函数：

int mm\_init(void)//初始化隐式空闲链表

{

if ((heap\_listp = mem\_sbrk(4 \* WSIZE)) == (void\*)-1)**//注意，mem\_sbrk为一个外部函数，位于memlib.c中，其作用为申请额外的堆内存，返回申请位置的首地址，此处是为了得到堆的起始位置，同时如果初始化失败，返回-1.**

return -1;

PUT(heap\_listp, 0);//将初始位置置零

PUT(heap\_listp + (1 \* WSIZE), PACK(DSIZE, 1));//将初始位置偏移量为一个字的位置设为双字大小，有效位为1

PUT(heap\_listp + (2 \* WSIZE), PACK(DSIZE, 1)); //将初始位置偏移量为两个字的位置设为双字大小，有效位为1

PUT(heap\_listp + (3 \* WSIZE), PACK(0, 1)); //将初始位置偏移量为三个字的位置设为0，有效位为1

heap\_listp += (2 \* WSIZE);//将heap\_listp偏移两个字

if (extend\_heap(CHUNKSIZE / WSIZE) == NULL)//extend\_heap函数见下面

return -1;//extend\_heap返回异常，初始化失败

return 0;//正常返回

}

Extend\_heap函数：

static void\* extend\_heap(size\_t words)//扩充堆，成功的话执行合并操作，并返回申请空间指向有效荷载的指针，否则返回NULL

{

char\* bp;

size\_t size;

size = (words % 2) ? (words + 1) \* WSIZE : words \* WSIZE;//将words对齐，并算出该字个数对应的字节数

if ((long)(bp = mem\_sbrk(size)) == -1)//调用mem\_sbrk函数提高堆空间

return NULL;//失败返回NULL

PUT(HDRP(bp), PACK(size, 0));//对新分配的块的头部进行初始化

PUT(FTRP(bp), PACK(size, 0)); //对新分配的块的脚部进行初始化

PUT(HDRP(NEXT\_BLKP(bp)), PACK(0, 1));//处理结尾块

return coalesce(bp);//调用合并函数（由于链表中最后一个块有可能为空闲块，故分配后有可能出现两个空闲块，故还需判断是否需要合并）

}

Coalesce函数：

static void\* coalesce(void\* bp)//将bp指向的块与前后空闲的块合并，并返回合并后的“有效荷载的”首地址

{

size\_t prev\_alloc = GET\_ALLOC(FTRP(PREV\_BLKP(bp)));//得到前一个块的分配情况

size\_t next\_alloc = GET\_ALLOC(HDRP(NEXT\_BLKP(bp)));//得到后一个块的分配情况

size\_t size = GET\_SIZE(HDRP(bp));//得到本块的大小

if (prev\_alloc && next\_alloc)//前后两块都已分配

{

**//跟原书的做法相比，该处我进行了一个小小的优化，能使分数高一分**

PUT(HDRP(bp), PACK(size, 0));//更新该块头部的状态，这一步和接下来的这一步是从mm\_free函数那儿贴过来的，当时分析这一步时有个疑问，其他的if else条件里都改了对应块的状态，就这个没改，后面分析到mm\_free函数时发现可以将这两个语句放到这里来，就试了一下，发现分数高了一分

PUT(FTRP(bp), PACK(size, 0));//更新该块脚部的状态

return bp; //一定要保留这句话，后面太多if else了，浪费时间

}

else if (prev\_alloc && !next\_alloc)//只有后块未分配

{

size += GET\_SIZE(HDRP(NEXT\_BLKP(bp)));//将两个块的大小之和拿到

PUT(HDRP(bp), PACK(size, 0));//更新bp的头部

PUT(FTRP(bp), PACK(size, 0));//再更新脚部（该处设计较为精妙，FTRP本身“调用”了HDRP，改完头部就可以直接改脚部了

}

else if (!prev\_alloc && next\_alloc)//只有前块未分配

{

size += GET\_SIZE(HDRP(PREV\_BLKP(bp)));//将两块大小之和拿到

PUT(FTRP(bp), PACK(size, 0));//先改脚部，因为bp相比于前块在后面

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));//此时不得不“调用”PREV\_BLKP得到前块地址，再来改前块的size大小

bp = PREV\_BLKP(bp);//将bp更新为前块

**//注意，此处并没有处理bp原始的头部和前块的脚部(上面一种情况和下面一种情况同理)，是因为合并后的块会直接跳到下一个块，该部分不再会被这个链表访问，即外面看是删除了，里面的数据实际还在，只不过无法到达**

}

else //都未分配，全部要合并

{

size += GET\_SIZE(HDRP(PREV\_BLKP(bp))) +

GET\_SIZE(FTRP(NEXT\_BLKP(bp)));//得到三块大小之和

PUT(HDRP(PREV\_BLKP(bp)), PACK(size, 0));//改前块的头部的大小

PUT(FTRP(NEXT\_BLKP(bp)), PACK(size, 0));//改后块的大小

bp = PREV\_BLKP(bp);//更新bp为前块

}

return bp;

}

mm\_alloc函数：

void\* mm\_malloc(size\_t size)//分配size大小的空间

{

size\_t asize;//实际要分配的空间(对齐要求)

size\_t extendsize;//堆要增长的大小

char\* bp;

if (size == 0)

return NULL;//分配空间为零，不分配

//下面这个if else是为了确定实际分配的大小

if (size <= DSIZE)

asize = 2 \* DSIZE;

else

asize = DSIZE \* ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);//size加上双字(头部和脚部)再加上补偿(类似于表示浮点数的那个补偿)除以双字，得到最少要多少个双字，最后乘以双字得到总的字节数

if ((bp = find\_fit(asize)) != NULL)//如果堆中有适配的空闲块，代码分析见place函数的下面

{

place(bp, asize);//该函数将大小为asize的块设为已占用，代码分析见下面

return bp;//返回分配好的指针

}

extendsize = MAX(asize, CHUNKSIZE);//未找到适配块，判断堆需要增长的大小

if ((bp = extend\_heap(extendsize / WSIZE)) == NULL)//如果增长堆失败

return NULL;

place(bp, asize);//成功增长后，将大小为asize的块设为已占用

return bp; //返回分配好的指针

}

Place函数：//自己实现的版本

static void place(void\* bp, size\_t asize)

{

size\_t whole\_size = GET\_SIZE(HDRP(bp));//whole\_size表示bp这个块的实际大小

if ((whole\_size - asize) >= (2 \* DSIZE))//判断是否要对该空闲块进行分割(asize中不包含头部和脚部，所以这里是2\*DSIZE,一个DSIZE是头部加脚部，另一个DSIZE是填充)

{

//要分割

PUT(HDRP(bp), PACK(asize, 0x1));//更新头部

PUT(FTRP(bp), PACK(asize, 0x1));//更新脚部

//该处的设计和合并函数coalesce中只有后块未分配的情况一样，利用FTRP“调用”HDRP的特性

bp = FTRP(bp) + DSIZE;//脚部加上一个双字(bp的脚部大小和下一个块的头部大小)，得到下一个块的bp

PUT(HDRP(bp), PACK(whole\_size - asize, 0x0));//更新头部

PUT(FTRP(bp), PACK(whole\_size - asize, 0x0));//更新脚部

}

else

{

//不用分割，直接更新即可

PUT(HDRP(bp), PACK(whole\_size, 0x1));//更新头部

PUT(FTRP(bp), PACK(whole\_size, 0x1));//更新脚部

}

}

Find\_fit函数：//自己实现的版本

static void\* find\_fit(size\_t asize)//该分配函数采用首次适配

{

void\* bp = heap\_listp;//从链表的头开始遍历

while(GET\_SIZE(HDRP(bp)) > 0)//当游走指针尚未到达结尾块时继续遍历(结尾块的意义！)

{

if (!GET\_ALLOC(HDRP(bp)) && (GET\_SIZE(HDRP(bp)) >= asize))//找到了适配的块(两个判断条件，是否已分配和大小是否足够)

return bp;

bp = NEXT\_BLKP(bp);//更新bp为下一个块

}

return NULL; //没有找到合适的，返回空指针

}

mm\_free函数：

void mm\_free(void\* ptr)

{

size\_t size = GET\_SIZE(HDRP(ptr));//得到该块的大小

//下面这两句在我的优化版本里是没有的，可以将这两句放coalesce里处理

PUT(HDRP(ptr), PACK(size, 0));//更新该块头部的状态

PUT(FTRP(ptr), PACK(size, 0));//更新该块脚部的状态

coalesce(ptr);//判断是否合并

}

mm\_realloc函数:

void\* mm\_realloc(void\* ptr, size\_t size)

{

void\* oldptr = ptr;//旧指针的位置

void\* newptr;//创建一个新指针

size\_t copySize;//复制内存的大小

newptr = mm\_malloc(size);//先开辟新区域

if (newptr == NULL)//无法分配

return NULL;

copySize = \*(size\_t\*)((char\*)oldptr - SIZE\_T\_SIZE);//拿到除头部和脚部的大小(有效荷载加填充)

if (size < copySize)//重新分配的位置更小

copySize = size;

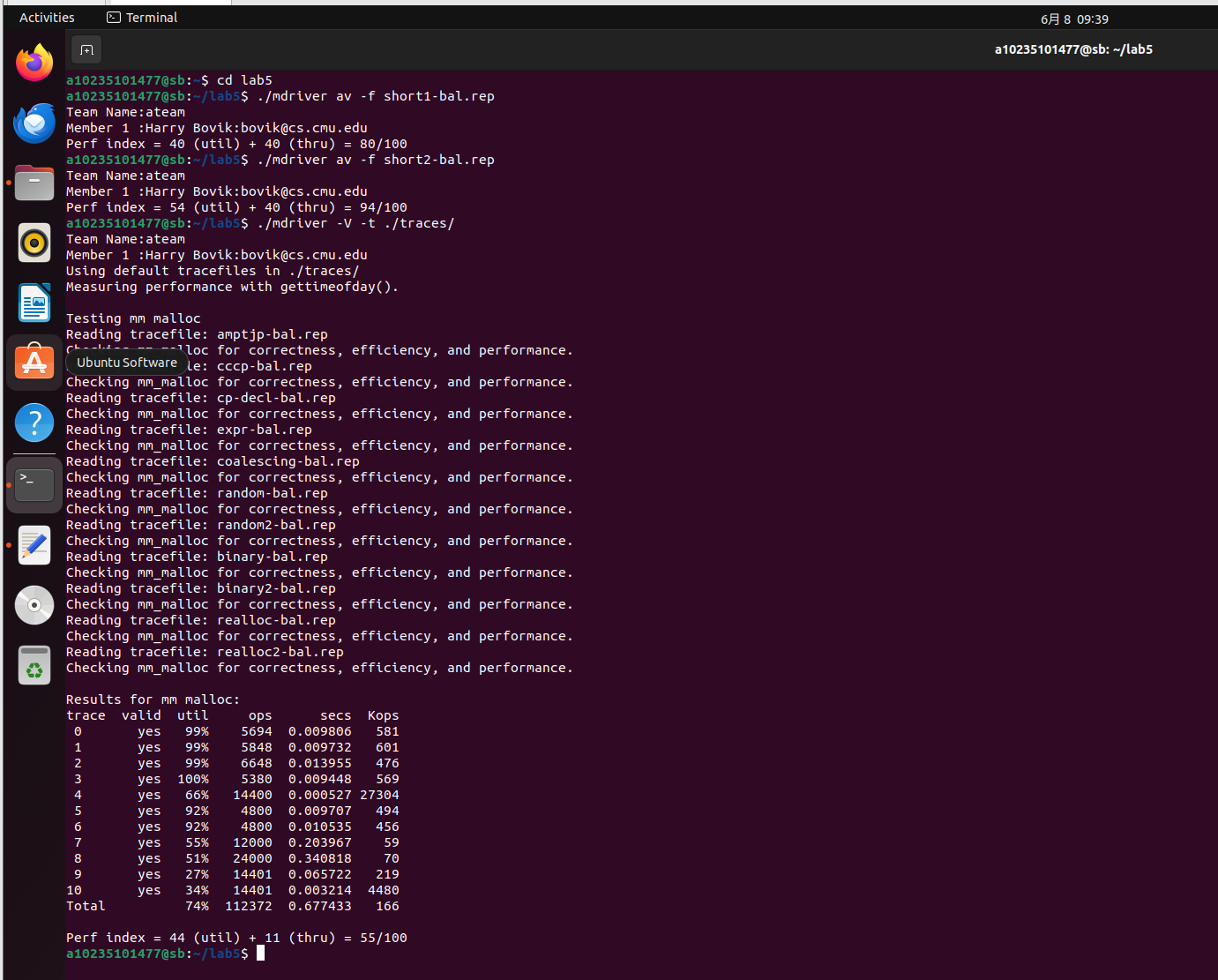
memcpy(newptr, oldptr, copySize);//将内容拷贝进去

mm\_free(oldptr);//释放旧指针

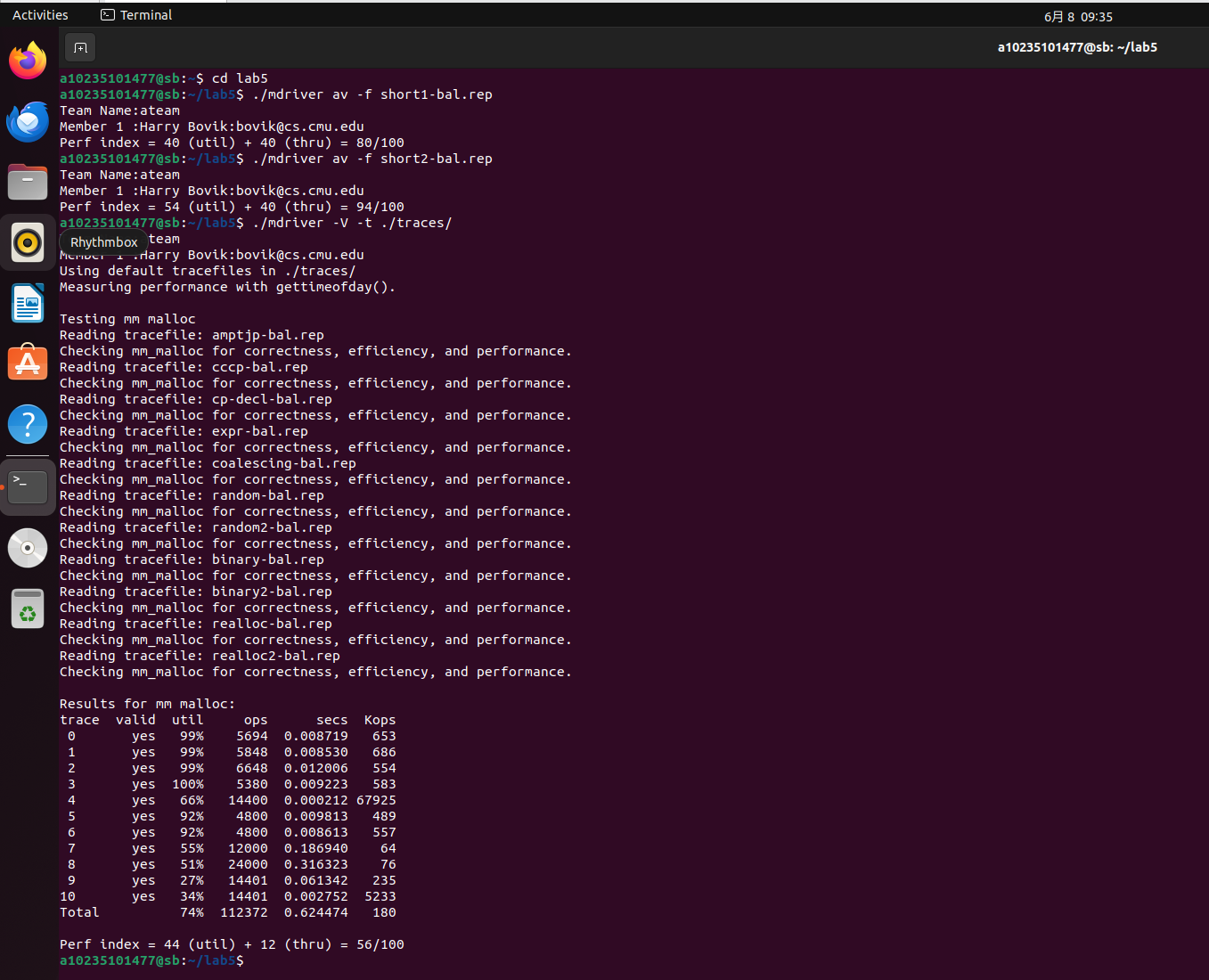
return newptr;//返回新指针

}

原书版本的跑分：



我优化版本的跑分：



后记：

本次实验涉及大量的指针操作，当时我想不用书上的宏做(相当于要接受他的思路)，但是当我发现在不能定义任何结构体时(不像实现cache那样)，我才感受到这个实验空前的难度，只有几个静态变量，还不能是数组(网上的有些高分做法违规了，使用了链表的数组实现分离适配)，后面结合时间的问题，便还是没有自己想出方案来，只是对原书的做法进行了自己的优化，虽然没有自己实现allocator，其中隐式空闲链表的实现方式是彻底理解了，包括设置序言块和结尾块的意义(当时我觉得设置这两块完全是多此一举，不过分析完代码后成小丑了)，此外，还增加了卡常的能力？那一分的增加是我完全没想到的，就试了一下，没想到确实快了。