重 庆 大 学

学 生 实 验 报 告

实验课程名称 操作系统

开课实验室 DS1502

学 院 大数据与软件学院 年级 22 专业班 软工 2 班

学 生 姓 名 潘铷葳 学 号 20221982

开 课 时 间 2023 至 2024 学年第 二 学期

总 成 绩	
教师签名	

软件学院制

《操作系统原理》实验报告

开课实验室： **2024 年 6 月 23 日**

学院	大数据与软件学院	年级、专业、班	2022 级软件工程 2 班	姓名	潘铷葳	成绩	
课程名称	操作系统	实验项目名称	实验五 内存管理	指导教师	刘寄		
教师评语	教师签名：刘寄 年 月 日						

1．实验目的：

（1）掌握内存分配器：

a）通过实现内存分配器，深入理解内存管理的基本原理和机制。

b）掌握内存分配和释放的基本操作，理解内存碎片及其处理方法。

（2）实现自己的 malloc/free：

a）自主实现内存分配和释放函数，提高对操作系统内存管理模块的理解和动手能力。

b）掌握动态内存分配的策略和算法，能够在实际项目中灵活运用。

2．实验内容：

1.实现内存分配算法：

实现首次适应（First-fit）、最佳适应（Best-fit）或最坏适应（Worst-fit）中的一种内存分配算法。

选择适当的数据结构和算法，提高内存分配的效率和利用率。

2.编辑 userapp/myalloc.c 文件：

实现如下四个接口函数：

void *malloc(size_t size)：分配指定大小的内存块。

void free(void *ptr)：释放指定内存块。

void *calloc(size_t num, size_t size)：分配并初始化指定数量和大小的内存块。

void *realloc(void *oldptr, size_t size)：重新分配指定内存块，调整大小。

3.接口函数实现：

tlsf_create_with_pool：

初始化内存分配器，设置初始内存块状态。

malloc：

分配大小为 size 字节的内存块，并返回块起始地址。

处理 size 为 0 的情况，返回 NULL。

free：

释放指定内存块，并合并相邻的空闲块。

验证指针的有效性，确保其指向有效的内存块。

calloc：

分配并初始化指定数量和大小的内存块，确保内存初始化为 0。

realloc：

重新分配内存块，并复制旧内存块的内容到新内存块。

处理 oldptr 为 NULL 或 size 为 0 的特殊情况。

4.线程安全：

使用信号量或互斥锁保护内存分配和释放的关键区域，确保多线程环境下的安全性。

5.测试和验证：

修改 userapp/Makefile 以启用自定义内存分配器。

在 main.c 中调用 test_allocator() 测试自定义内存分配器的功能和性能。

3．实验报告：

myalloc.c文件

```c
/**
 * vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4
 */
#include <sys/types.h>
#include <string.h>
#include <stdint.h>

struct chunk {
    char signature[4];  /* "OSEX" */
    struct chunk *next; /* ptr. to next chunk */
    int state;          /* 0 - free, 1 - used */
#define FREE   0
#define USED   1

    int size;           /* size of this chunk */
};

static struct chunk *chunk_head;

void init_memory_pool(size_t heap_size, uint8_t *heap_base)
{
    chunk_head = (struct chunk *)heap_base;
    strncpy(chunk_head->signature, "OSEX", 4);
    chunk_head->next = NULL;
    chunk_head->state = FREE;
    chunk_head->size  = heap_size;
}

//实验五
void* g_heap;
void* tlsf_create_with_pool(uint8_t* heap_base, size_t heap_size)
{
    chunk_head = (struct chunk*)heap_base;
    strncpy(chunk_head->signature,"OSEX", 4);
    chunk_head->next = NULL;
    chunk_head->state = FREE;
    chunk_head->size = heap_size;
    return NULL;
}
struct chunk add_block(int size) {
    struct chunk* a;
```

```
        struct chunk b;
        a = &b;
        a->state = FREE;
        a->size = size;
        a->next = NULL;
        strncpy(a->signature, "OSEX", 4);
        return b;
    }
    struct chunk* findFreeblock(size_t size) {
        struct chunk* a = chunk_head;
        while (a != NULL) {
            if (a->size >= size && a->state == FREE) {
                return a;
            }
            a = a->next;
        }
        return NULL;
    }
    void *malloc(size_t size)//分配大小为 size 字节的内存块，并返回块起始地址，如果 size 是 0，返回 NULL
    {
        if (size != 0) {
            struct chunk* tmp = findFreeblock(size);
            if (tmp == NULL)return NULL;
            else {
                void* ptr;
                if (strncmp(tmp->signature, "OSEX", 4) == 0)
                    ptr = (uint8_t*)tmp + sizeof(struct chunk);
                else
                    return NULL;
                int freesize= tmp->size - size - sizeof(struct chunk);
                if (freesize <= 0) {
                    tmp->state = USED;
                    return ptr;
                }
                else {
                    struct chunk* new = (struct chunk*)((uint8_t*)ptr + size);
                    *new = add_block(freesize);
                    tmp->size = size;
                    tmp->state = USED;
                    new->next = tmp->next;
                    tmp->next = new;
                    return ptr;//相当于插入链表
```

```
84.
85.                 }
86.             }
87.         }
88.         else
89.         {
90.             return NULL;
91.         }
92.     }
93.
94.     void free(void *ptr)//释放 ptr 指向的内存块,如果 ptr 是 NULL，直接返回
95.     {
96.         if (ptr != NULL) {
97.             struct chunk* target_free = (struct chunk*)(((uint8_t*)ptr) - sizeof(struct chunk));
98.              if (strncmp(target_free->signature, "OSEX", 4) != 0)//验证签名
99.                 return;
100.             if (target_free != NULL) {                              //进行合并
101.                 target_free->state = FREE;
102.                 struct chunk* curr_chunk = chunk_head;
103.                 while (curr_chunk != NULL) {
104.                     struct chunk* curr_next = curr_chunk->next;
105.                     if (curr_chunk->state == FREE) {
106.                         size_t size = curr_chunk->size;
107.                         while (curr_next->state == FREE && curr_next != NULL) {
108.                             size = size + (curr_next->size + sizeof(struct chunk));
109.                              curr_next = curr_next->next;          //其实只要进入该代码一次即可，可以设置 flag 判断有无进入。
110.                                                                   //也可以先找到 target_free 的前一个节点和后一个节点，对他们进行判断合并。
111.                         }
112.                         curr_chunk->next = curr_next;
113.                         curr_chunk->size = size;
114.                     }
115.                     curr_chunk = curr_next;
116.                 }
117.             }
118.         }
119.         else
120.             return;
```

```
121.  }
122.
123.  void *calloc(size_t num, size_t size)//为 num 个元素的数组分配内存，每个元
    素占 size 字节，把分配的内存初始化成 0
124.  {
125.      size_t sizetotal = num * size;
126.      void* ptr = malloc(sizetotal);
127.      if (ptr == NULL) {
128.          return NULL;
129.      }
130.      else {
131.          size_t i;
132.          for (i = 0; i < sizetotal; i++)
133.          {
134.              *((uint8_t*)ptr + i) = 0;
135.          }
136.          return ptr;
137.      }
138.  }
139.
140.  //重新分配 oldptr 指向的内存块，新内存块有 size 字节
141.  //如果 oldptr 是 NULL，该函数等价于 malloc(size)
142.  //如果 size 是 0，该函数等价于 free(oldptr)
143.  //把旧内存块的内容复制到新内存块
144.  //如果新内存块比较小，只复制旧内存块的前面部分
145.  //如果新内存块比较大，复制整个旧内存块，而且不用初始化多出来的那部分
146.  //如果新内存块还在原来的地址 oldptr，返回 oldptr；否则返回新地址
147.
148.  //必须验证 oldptr 的有效性
149.  //必须合并相邻的空闲块
150.
151.  void *realloc(void *oldptr, size_t size)
152.  {
153.      if (size != 0) {
154.          void* ptr = malloc(size);//先分配一块空间
155.          if (oldptr == NULL)return ptr;
156.          else {
157.              struct chunk* oldchunk = (struct chunk*)(((uint8_t*)ptr) -
    sizeof(struct chunk));//找到块头
158.              if (strncmp(oldchunk->signature, "OSEX", 4) != 0)return NU
    LL;
159.              if (oldchunk != NULL) {
160.                  int curr_size;
161.                  if (oldchunk->size > size) {
```

```
                    curr_size = size;
                }
                else {
                    curr_size = oldchunk->size;
                }
                int i;
                for (i = 0; i < curr_size; i++) {
                    *((uint8_t*)ptr + i) = *((uint8_t*)oldptr + i);
                }
                free(oldptr);
                return ptr;

            }
        }
        return ptr;
    }
    else {
        free(oldptr);
        return NULL;
    }
}
```

```
COBJS=  vm86call.o graphics.o main.o
COBJS+= lib/sysconf.o lib/math.o lib/stdio.o lib/stdlib.o \
        lib/qsort.o
COBJS+= ../lib/softfloat.o ../lib/string.o ../lib/memcpy.o \
        ../lib/memset.o ../lib/snprintf.o
COBJS+= myalloc.o
```

进行测试

Step1:在 myalloc.c 中增加头文件：

```
#include <unistd.h>
#include <syscall.h>
#include <stdio.h>
```

Step2：在 myalloc.c 中直接编写 test_alloc()函数

```
static void tsk_malloc(void* pv)
{
    int i, c = (int)pv;
    char** a = malloc(c * sizeof(char*));
    for (i = 0; i < c; i++) {
        a[i] = malloc(i + 1);
        a[i][i] = 17;
    }
    for (i = 0; i < c; i++) {
        free(a[i]);
```

```
        }
        free(a);
        task_exit(0);
    }
```

Step3：test_allocator()函数的代码如下

```
#define MESSAGE(foo) printf("%s, line %d: %s", __FILE__, __LINE__, foo)
void test_allocator()
{
    char* p, * q, * t;

    MESSAGE("[1] Test malloc/free for unusual situations\r\n");

    MESSAGE("[1.1] Allocate small block ... ");
    p = malloc(17);
    if (p == NULL) {
        printf("FAILED\r\n");
        return;
    }
    p[0] = p[16] = 17;
    printf("PASSED\r\n");

    MESSAGE("[1.2] Allocate big block ... ");
    q = malloc(4711);
    if (q == NULL) {
        printf("FAILED\r\n");
        return;
    }
    q[4710] = 47;
    printf("PASSED\r\n");

    MESSAGE("[1.3] Free big block ... ");
    free(q);
    printf("PASSED\r\n");

    MESSAGE("[1.4] Free small block ... ");
    free(p);
    printf("PASSED\r\n");

    MESSAGE("[1.5] Allocate huge block ... ");
    q = malloc(32 * 1024 * 1024 - sizeof(struct chunk));
    if (q == NULL) {
        printf("FAILED\r\n");
        return;
```

```
39.         }
40.         q[32 * 1024 * 1024 - sizeof(struct chunk) - 1] = 17;
41.         free(q);
42.         printf("PASSED\r\n");
43.
44.         MESSAGE("[1.6] Allocate zero bytes ... ");
45.         if ((p = malloc(0)) != NULL) {
46.             printf("FAILED\r\n");
47.             return;
48.         }
49.         printf("PASSED\r\n");
50.
51.         MESSAGE("[1.7] Free NULL ... ");
52.         free(p);
53.         printf("PASSED\r\n");
54.
55.         MESSAGE("[1.8] Free non-allocated-via-malloc block ... ");
56.         int arr[5] = { 0x55aa4711, 0x5aa5a1147, 0xa5aa51471, 0xaa551471, 0x5a
    a54171 };
57.         free(arr);
58.         if (arr[0] == 0x55aa4711 && arr[1] == 0x5aa5a1147 && arr[2] == 0xa5aa
    51471 && arr[3] == 0xaa551471 && arr[4] == 0x5aa54171) {
59.             printf("PASSED\r\n");
60.         }
61.         else {
62.             printf("FAILED\r\n");
63.             return;
64.         }
65.
66.         MESSAGE("[1.9] Various allocation pattern ... ");
67.         int k;
68.         size_t pagesize = sysconf(_SC_PAGESIZE);
69.         for (i = 0; i < 7411; i++) {
70.             p = malloc(pagesize);
71.             p[pagesize - 1] = 17;
72.             q = malloc(pagesize * 2 + 1);
73.             q[pagesize * 2] = 17;
74.             t = malloc(1);
75.             t[0] = 17;
76.             free(p);
77.             free(q);
78.             free(t);
79.         }
80.
```

```
81.         char** a = malloc(2741 * sizeof(char*));
82.         for (i = 0; i < 2741; i++) {
83.             a[i] = malloc(i + 1);
84.             a[i][i] = 17;
85.         }
86.         for (i = 0; i < 2741; i++) {
87.             free(a[i]);
88.         }
89.         free(a);
90.
91.         if (chunk_head->next != NULL || chunk_head->size != 32 * 1024 * 1024)
    {
92.             printf("FAILED\r\n");
93.             return;
94.         }
95.         printf("PASSED\r\n");
96.
97.         MESSAGE("[1.10] Allocate using calloc ... ");
98.         int* x = calloc(17, 4);
99.         for (i = 0; i < 17; i++) {
100.            if (x[i] != 0) {
101.                printf("FAILED\r\n");
102.                return;
103.            }
104.            else {
105.                x[i] = i;
106.            }
107.        }
108.        free(x);
109.        printf("PASSED\r\n");
110.
111.        MESSAGE("[2] Test realloc() for unusual situations\r\n");
112.
113.        MESSAGE("[2.1] Allocate 17 bytes by realloc(NULL, 17) ... ");
114.        p = realloc(NULL, 17);
115.        if (p == NULL) {
116.            printf("FAILED\r\n");
117.            return;
118.        }
119.        p[0] = p[16] = 17;
120.        printf("PASSED\r\n");
121.
122.        MESSAGE("[2.2] Increase size by realloc(..., 4711) ... ");
123.        p = realloc(p, 4711);
```

```
124.     if (p == NULL) {
125.         printf("FAILED\r\n");
126.         return;
127.     }
128.     if (p[0] != 17 || p[16] != 17) {
129.         free(p);
130.         printf("FAILED\r\n");
131.         return;
132.     }
133.     p[4710] = 47;
134.     printf("PASSED\r\n");
135. 
136.     MESSAGE("[2.3] Decrease size by realloc(..., 17) ... ");
137.     p = realloc(p, 17);
138.     if (p == NULL) {
139.         printf("FAILED\r\n");
140.         return;
141.     }
142.     if (p[0] != 17 || p[16] != 17) {
143.         free(p);
144.         printf("FAILED\r\n");
145.         return;
146.     }
147.     printf("PASSED\r\n");
148. 
149.     MESSAGE("[2.4] Free block by realloc(..., 0) ... ");
150.     p = realloc(p, 0);
151.     if (p != NULL) {
152.         printf("FAILED\r\n");
153.         return;
154.     }
155.     else {
156.         printf("PASSED\r\n");
157.     }
158. 
159.     MESSAGE("[2.5] Free block by realloc(realloc(NULL, 0), 0) ... ");
160.     p = realloc(realloc(NULL, 0), 0);
161.     if (p != NULL) {
162.         printf("FAILED\r\n");
163.         return;
164.     }
165.     else {
166.         printf("PASSED\r\n");
167.     }
```

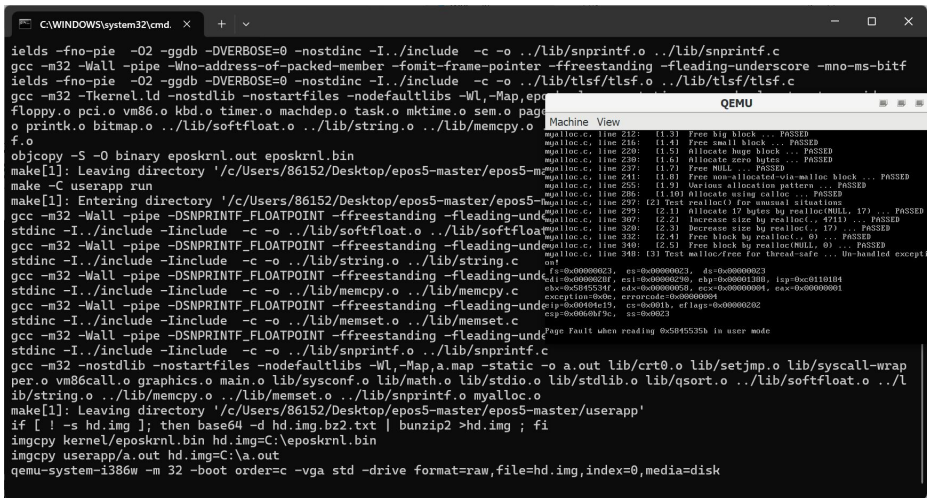
```
168.
169.      MESSAGE("[3] Test malloc/free for thread-safe ... ");
170.      int t1, t2;
171.      char* s1 = malloc(1024 * 1024);
172.      char* s2 = malloc(1024 * 1024);
173.      task_create(&t1, tsk_malloc, (void*)5000);
174.      task_create(&t2, tsk_malloc, (void*)5000);
175.      task_wait(t1, NULL);
176.      task_wait(t2, NULL);
177.      free(s1);
178.      free(s2);
179.
180.      if (chunk_head->next != NULL || chunk_head->size != 32 * 1024 * 1024)
    {
181.          printf("FAILED\r\n");
182.          return;
183.      }
184.      printf("PASSED\r\n");
185.  }
```

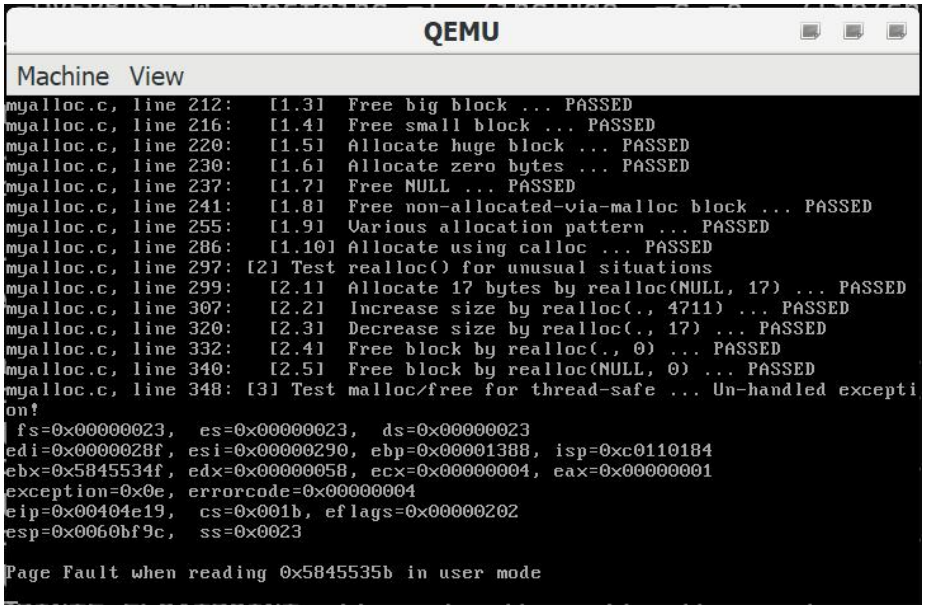
Main 主函数如下：

```
1.   /*
2.    * vim: filetype=c:fenc=utf-8:ts=4:et:sw=4:sts=4
3.    */
4.   #include <inttypes.h>
5.   #include <stddef.h>
6.   #include <math.h>
7.   #include <stdio.h>
8.   #include <sys/mman.h>
9.   #include <syscall.h>
10.  #include <netinet/in.h>
11.  #include <stdlib.h>
12.  #include "graphics.h"
13.  #include<time.h>
14.
15.  extern void* tlsf_create_with_pool(void* mem, size_t bytes);
16.  extern void* g_heap;
17.
18.  /**
19.   * GCC insists on __main
20.   *    http://gcc.gnu.org/onlinedocs/gccint/Collect2.html
21.   */
22.  void __main()
23.  {
```

```
24.         size_t heap_size = 32 * 1024 * 1024;
25.         void* heap_base = mmap(NULL, heap_size, PROT_READ | PROT_WRITE, MAP_P
    RIVATE | MAP_ANON, -1, 0);
26.         g_heap = tlsf_create_with_pool(heap_base, heap_size);
27.     }
28.
29.     void main(void* pv)
30.     {
31.
32.         extern void test_allocator();
33.     test_allocator();
34.     sleep(1000);
35.         while (1);
36.         task_exit(0);
37.     }
```

4. 运行结果：





5. 实验总结：

通过本次实验，我深入学习了内存分配器的实现原理和具体实现方法。以下是对实验过程和收获的总结：

（1）掌握内存分配器的工作原理：

了解了内存分配算法的基本思想和实现方法，特别是首次适应、最佳适应和最坏适应算法的区别和应用场景。

通过实现 malloc、free、calloc 和 realloc 函数，掌握了动态内存分配和释放的基本操作。

（2）内存管理的数据结构：

学习了内存块的数据结构 struct chunk，理解了内存块的管理方式和连接方式。

通过实现内存块的分配和释放函数，理解了内存碎片的产生和处理方法。

（3）调试和优化：

在实验过程中遇到了一些问题，如代码插入位置错误、编译错误、链接错误等，通过不断调试和优化，解决了这些问题。

使用 make clean 清除编译环境，确保每次编译结果的准确性和一致性。

（4）线程安全的实现：

在实现内存分配器时，考虑了多线程环境下的线程安全问题，使用信号量保护临界区，确保内存分配和释放操作的原子性。

（5）测试和验证：

修改 Makefile 并在 main.c 中测试自定义内存分配器，验证了内存分配器的功能和性能。