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 DS1503

学 院 大数据与软件学院 年级 软件工程 专业班 01

学 生 姓 名 学 号

开 课 时 间 2024 至 2025 学年第 二 学期

总 成 绩	
教师签名	

软件学院制

《操作系统》实验报告

开课实验室： **2025 年 3 月 1 日**

学院	大数据与软件学院	年级、专业、班	2023/软件工程/01	姓名		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： 年 月 日						

一、实验目的

实现线程创建和简单图形化实例

二、实验内容

- 实现首次/最佳/最坏中的一种分配算法
- 编辑文件userapp/myalloc.c，实现如下四个接口函数
  - malloc/free/calloc/realloc

四个函数要求：

- void *malloc(size_t size);
  - 功能
    - 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

void free(void *ptr);

- 功能
  - 释放*ptr*指向的内存块
  - 如果ptr是NULL，直接返回
- 提示
  - 怎么根据*ptr*得到chunk？
    - struct trunk *achunk=(struct chunk *)(((uint8_t *)*ptr*)-sizeof(struct chunk));
- 要求
  - 必须验证*ptr*的有效性
    - 判断achunk->signature是否等于“OSEX”
  - 必须合并相邻的空闲块

void *calloc(size_t num, size_t size);

- 功能
  - 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

void *realloc(void *oldptr, size_t size);

- 功能
  - 重新分配*oldptr*指向的内存块，新内存块有*size*字节
    - 如果*oldptr*是NULL，该函数等价于malloc(size)
    - 如果size是0，该函数等价于free(oldptr)
  - 把旧内存块的内容复制到新内存块
    - 如果新内存块比较小，只复制旧内存块的前面部分
    - 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
  - 如果新内存块还在原来的地址*oldptr*，*返回oldptr；否则返回新地址*
- 要求
  - 必须验证*oldptr*的有效性
  - 必须合并相邻的空闲块

线程安全要求：

Thread-safe

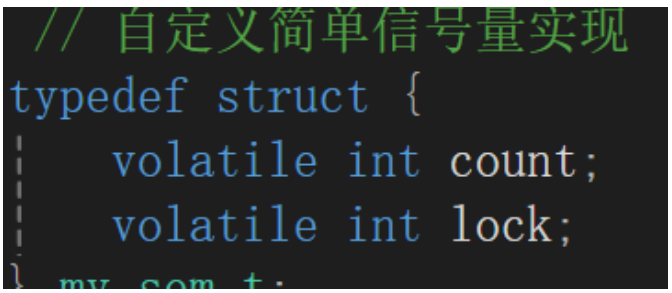
- Implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.
- 完善你的内存分配器，确保线程安全
  - 提示：用信号量保护临界区

三、使用仪器、材料

虚拟机，编译器

三、 实验过程原始记录(数据、图表、计算等)

实现首次分配算法，自定义了信号量实现



```
// 首次适应搜索
while (*pp) {
    struct chunk* cur = *pp;
    if (cur->state == FREE && cur->size >= needed) {
        // 分割内存块
        if (cur->size - needed >= CHUNK_SIZE) {
            struct chunk* new_chunk = (struct chunk*)((char*)cur + needed);
            memcpy(new_chunk->signature, "OSEX", 4);
            new_chunk->size = cur->size - needed;
            new_chunk->state = FREE;
            new_chunk->next = cur->next;

            cur->next = new_chunk;
            cur->size = needed;
        }
        cur->state = USED;
        result = (char*)cur + CHUNK_SIZE;
        break;
    }
    pp = &cur->next;
}
```

五、实验结果及分析

```
myalloc.c, line 240: [1] Test malloc/free for unusual situations
myalloc.c, line 242:   [1.1]  Allocate small block ... PASSED
myalloc.c, line 251:   [1.2]  Allocate big block ... PASSED
myalloc.c, line 260:   [1.3]  Free big block ... PASSED
myalloc.c, line 264:   [1.4]  Free small block ... PASSED
myalloc.c, line 268:   [1.5]  Allocate huge block ... PASSED
myalloc.c, line 278:   [1.6]  Allocate zero bytes ... PASSED
myalloc.c, line 285:   [1.7]  Free NULL ... PASSED
myalloc.c, line 289:   [1.8]  Free non-allocated-via-malloc block ... PASSED
myalloc.c, line 303:   [1.9]  Various allocation pattern ... PASSED
myalloc.c, line 334:   [1.10] Allocate using calloc ... PASSED
myalloc.c, line 345: [2] Test realloc() for unusual situations
myalloc.c, line 347:   [2.1]  Allocate 17 bytes by realloc(NULL, 17) ... PASSED
myalloc.c, line 355:   [2.2]  Increase size by realloc(., 4711) ... PASSED
myalloc.c, line 368:   [2.3]  Decrease size by realloc(., 17) ... PASSED
myalloc.c, line 380:   [2.4]  Free block by realloc(., 0) ... PASSED
myalloc.c, line 388:   [2.5]  Free block by realloc(NULL, 0) ... PASSED
myalloc.c, line 396: [3] Test malloc/free for thread-safe ... PASSED
```

userapp/myalloc.c 源码：

```
// 自定义简单信号量实现
typedef struct {
    volatile int count;
    volatile int lock;
} my_sem_t;

#define FREE 0
#define USED 1
#define CHUNK_SIZE sizeof(struct chunk)

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

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

```
};

static struct chunk *chunk_head=NULL;
static my_sem_t alloc_sem = { 1, 0 }; // 初始化为可用状态
// 原子交换操作（模拟 CAS）
static int atomic_swap(volatile int* ptr, int new) {
    int old;
    __asm__ __volatile__(
        "xchgl %0, %1"
        : "=r"(old), "+m"(*ptr)
        : "0"(new)
        : "memory"
    );
    return old;
}

// 信号量等待（P 操作）
static void sem_wait(my_sem_t* s) {
    while (1) {
        while (s->count == 0); // 忙等待
        if (atomic_swap(&s->lock, 1) == 0) {
            if (s->count > 0) {
                s->count--;
                atomic_swap(&s->lock, 0);
                return;
            }
            atomic_swap(&s->lock, 0);
        }
    }
}

// 信号量释放（V 操作）
static void sem_post(my_sem_t* s) {
    while (atomic_swap(&s->lock, 1));
    s->count++;
    atomic_swap(&s->lock, 0);
}

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;
}

// 内部合并实现（假设已持有锁）
static void coalesce(struct chunk* chunk) {
    // 前向合并
    if (chunk != chunk_head) {
        struct chunk* prev = chunk_head;
        while (prev->next != chunk) prev = prev->next;
        if (prev->state == FREE && (char*)prev + prev->size == (char*)chunk) {
            prev->size += chunk->size;
            prev->next = chunk->next;
            chunk = prev;   // 更新当前块为合并后的块
        }
    }

    // 后向合并
    struct chunk* next = chunk->next;
    if (next && next->state == FREE && (char*)chunk + chunk->size == (char*)next) {
        chunk->size += next->size;
        chunk->next = next->next;
    }
}

void *malloc(size_t size)
{
    if (size == 0 ) return NULL;

    sem_wait(&alloc_sem); // 进入临界区

    struct chunk** pp = &chunk_head;
    size_t needed = size + CHUNK_SIZE;
    void* result = NULL;

    // 首次适应搜索
    while (*pp) {
        struct chunk* cur = *pp;
        if (cur->state == FREE && cur->size >= needed) {
            // 分割内存块
            if (cur->size - needed >= CHUNK_SIZE) {
                struct chunk* new_chunk = (struct chunk*)((char*)cur + needed);
                memcpy(new_chunk->signature, "OSEX", 4);
                new_chunk->size = cur->size - needed;
                new_chunk->state = FREE;
```

```
                    new_chunk->next = cur->next;

                    cur->next = new_chunk;
                    cur->size = needed;
                }
                cur->state = USED;
                result = (char*)cur + CHUNK_SIZE;
                break;
            }
            pp = &cur->next;
        }

        sem_post(&alloc_sem); //  离开临界区
        return result;
}

void free(void *ptr)
{
        if (!ptr) return;

       sem_wait(&alloc_sem);    //  进入临界区

        struct chunk* chunk = (struct chunk*)((uint8_t*)ptr - CHUNK_SIZE);
        if (strncmp(chunk->signature, "OSEX", 4) == 0) {
            chunk->state = FREE;
            coalesce(chunk);    //  合并相邻空闲块
        }

        sem_post(&alloc_sem); //  离开临界区
}

void *calloc(size_t num, size_t size)
{
        size_t total = num * size;
        if (total == 0 ) return NULL;

        void* ptr = malloc(total);
        if (ptr) memset(ptr, 0, total);
        return ptr;
}

void *realloc(void *oldptr, size_t size)
{
        if (!oldptr) return malloc(size);
        if (size == 0) {
            free(oldptr);
```

```
        return NULL;
    }

    sem_wait(&alloc_sem); // 进入临界区

    void* newptr = NULL;
    struct chunk* chunk = (struct chunk*)((uint8_t*)oldptr - CHUNK_SIZE);

    if (strncmp(chunk->signature, "OSEX", 4) != 0) {
        sem_post(&alloc_sem);
        return NULL;
    }

    size_t needed = size + CHUNK_SIZE;

    // 尝试就地扩展
    if (chunk->size >= needed) {
        // 检查是否需要分割
        if (chunk->size - needed >= CHUNK_SIZE) {
            struct chunk* new_chunk = (struct chunk*)((char*)chunk + needed);
            memcpy(new_chunk->signature, "OSEX", 4);
            new_chunk->size = chunk->size - needed;
            new_chunk->state = FREE;
            new_chunk->next = chunk->next;

            chunk->next = new_chunk;
            chunk->size = needed;
        }
        newptr = oldptr;
    }
    else {
        // 尝试合并后方空间
        coalesce(chunk);
        if (chunk->size >= needed) {
            newptr = oldptr;
        }
        else {
            // 分配新空间并迁移数据
            newptr = malloc(size);
            if (newptr) {
                size_t old_size = chunk->size - CHUNK_SIZE;
                memcpy(newptr, oldptr, old_size < size ? old_size : size);
                // 直接标记旧块为 FREE 并合并，避免调用 free()
                chunk->state = FREE;
                coalesce(chunk);
            }
```

```
        }
    }

    sem_post(&alloc_sem); // 离开临界区
    return newptr;
}
```

实验报告打印格式说明

1. 标题：三号加粗黑体
2. 开课实验室：5 号加粗宋体
3. 表中内容：

(1) 标题：5 号黑体

(2) 正文：5 号宋体

4. 纸张：16 开(20cm×26.5cm)
5. 版芯

上距：2cm

下距：2cm

左距：2.8cm

右距：2.8cm

说明： 1、“年级专业班”可填写为“00 电子 1 班”，表示 2000 级电子工程专业第 1 班。

2、实验成绩可按五级记分制（即优、良、中、及格、不及格），或者百分制记载，若需要将实验成绩加入对应课程总成绩的，则五级记分应转换为百分制。