# 内存分配性能和碎片化测试模块

## 概述

本模块实现了一个综合的内存分配性能测试系统，用于评估内存管理系统的分配效率和碎片化情况。测试通过生成所有可能的malloc/free操作序列，使用钩子函数监控内存分配过程，收集和分析内存分配时间、内部碎片和外部碎片的统计数据。

## 核心功能

### 1. 内部碎片检测

* **检测机制**：通过钩子函数比较实际申请大小与系统分配大小的差异
* **测量指标**：内部碎片率 = (申请大小 - 实际分配大小) / 申请大小

### 2. 外部碎片检测

* **检测机制**：定时查询系统中最大可用内存块大小
* **测量指标**：外部碎片率 = 最大可用块大小 / 总堆内存大小

### 3. 内存分配性能测量

* **测量对象**：malloc和free操作的执行时间
* **性能指标**：平均分配时间、平均释放时间

## 设计思路

### 1. 内存碎片的测量

内存碎片分为内部碎片和外部碎片两种：

* **内部碎片**：由于内存对齐或分配器策略导致的单个分配块内部的浪费空间
* **外部碎片**：由于频繁的分配和释放导致的内存空间不连续，无法满足大块内存分配需求

本模块通过钩子函数在内存分配时记录申请大小和实际分配大小，计算内部碎片率。通过定期查询最大可用块来评估外部碎片情况。

### 2. Catalan数序列生成

为了全面测试各种内存分配模式，本模块使用Catalan数生成所有可能的有效malloc/free操作序列：

int calculate\_catalan\_number(int n){  
 if(n == 0){  
 return 1;  
 }  
 return calculate\_catalan\_number(n - 1) \* (4 \* n - 2) / (n + 1);  
}

Catalan数确保了每个malloc操作都有对应的free操作，且遵循正确的嵌套关系。

### 3. 钩子函数监控

使用内存分配钩子函数实时监控内存操作：

void memory\_hook\_malloc(void\* ptr, int size){  
 u64 malloc\_time = sys\_timestamp();  
 // 记录分配时间和大小信息  
}  
  
void memory\_hook\_free(void\* ptr){  
 u64 free\_time = sys\_timestamp();  
 // 记录释放时间  
}

### 4. 栈结构管理

使用栈结构来管理分配的内存指针，确保LIFO(Last In First Out)的释放顺序：

typedef struct {  
 int items[MALLOC\_NUM]; // 存储统计项索引  
 int top; // 栈顶指针  
} Stack;

## 配置参数

#define MALLOC\_NUM 4 // 同时分配的内存块数量

## 代码结构

### 主要数据结构

// 统计数据结构  
typedef struct{  
 u64 memory\_malloc\_size; // 申请的内存大小  
 u64 memory\_allocate\_size; // 实际分配的内存大小  
 u64 memory\_malloc\_cost; // malloc操作耗时  
 u64 memory\_free\_cost; // free操作耗时  
 double external\_fragmentation\_rate; // 外部碎片率  
} statistics\_t;  
  
// 统计项结构  
typedef struct {  
 statistics\_t statistics; // 统计数据  
 void\* ptr; // 分配的内存指针  
} statistics\_item\_t;  
  
// 栈结构用于管理内存指针  
typedef struct {  
 int items[MALLOC\_NUM]; // 存储统计项索引  
 int top; // 栈顶指针  
} Stack;

### 核心函数

#### 1. generateMallocFreeSequence()

**功能**：递归生成所有有效的malloc/free操作序列

**参数**：

* malloc\_count: 当前malloc操作次数
* free\_count: 当前free操作次数
* sequence: 当前序列字符串
* sequences: 存储所有有效序列的二维数组

**实现逻辑**：

void generateMallocFreeSequence(int malloc\_count, int free\_count, char\* sequence,   
 int index, char\*\* sequences, int\* seq\_index) {  
 // 递归终止条件：malloc和free操作都达到指定数量  
 if (malloc\_count == MALLOC\_NUM && free\_count == MALLOC\_NUM) {  
 // 保存有效序列  
 sequences[\*seq\_index] = (char\*)malloc((index + 1) \* sizeof(char));  
 strcpy(sequences[\*seq\_index], sequence);  
 (\*seq\_index)++;  
 return;  
 }  
   
 // 如果malloc次数未达到上限，可以继续malloc  
 if (malloc\_count < MALLOC\_NUM) {  
 sequence[index] = 'M';  
 generateMallocFreeSequence(malloc\_count + 1, free\_count, sequence, index + 1, sequences, seq\_index);  
 }  
   
 // 如果free次数小于malloc次数，可以进行free操作  
 if (free\_count < malloc\_count) {  
 sequence[index] = 'F';  
 generateMallocFreeSequence(malloc\_count, free\_count + 1, sequence, index + 1, sequences, seq\_index);  
 }  
}

#### 2. memory\_hook\_malloc() 和 memory\_hook\_free()

**功能**：内存分配和释放的钩子函数

**malloc钩子实现**：

void memory\_hook\_malloc(void\* ptr, int size){  
 memset(&mem\_info, 0, sizeof(meminfo\_t));  
 u64 malloc\_time = sys\_timestamp();  
 if(mem\_getinfo(&mem\_info) == 0){  
 // 计算实际分配大小  
 statistics[statistics\_index].statistics.memory\_allocate\_size =   
 mem\_info.heapmemused - statistics[statistics\_index].statistics.memory\_allocate\_size;  
 // 记录分配时间  
 statistics[statistics\_index].statistics.memory\_malloc\_cost =   
 malloc\_time - statistics[statistics\_index].statistics.memory\_malloc\_cost;  
 }  
}

#### 3. 栈操作函数

**栈初始化**：

void init\_stack(Stack\* stack) {  
 stack->top = -1; // 栈顶指针初始化为-1，表示空栈  
}

**入栈操作**：

void push(Stack\* stack, int index) {  
 if (is\_full(stack)) {  
 printk("Stack is full, cannot push!\n");  
 return;  
 }  
 stack->items[++stack->top] = index; // 将索引压入栈  
}

**出栈操作**：

int pop(Stack\* stack) {  
 if (is\_empty(stack)) {  
 printk("Stack is empty, cannot pop!\n");  
 return -1;  
 }  
 return stack->items[stack->top--]; // 从栈中弹出索引  
}

#### 4. test\_memory\_allocate()

**功能**：主测试函数

**执行流程**：

1. **序列生成**：

* int catalan\_number = calculate\_catalan\_number(MALLOC\_NUM);  
  char\*\* sequences = (char\*\*)malloc(catalan\_number \* sizeof(char\*));  
  generateMallocFreeSequence(0, 0, sequence, 0, sequences, &seq\_index);

1. **钩子函数注册**：

* mem\_stat\_hook\_add(memory\_hook\_malloc, memory\_hook\_free);

1. **序列执行**：

* for (int i = 0; i < seq\_index; i++) {  
   Stack stack;  
   init\_stack(&stack);  
   char\* sequence = sequences[i];  
    
   // 执行序列中的每个操作  
   for(int j = 0; j < 2\*MALLOC\_NUM; j++){  
   if(sequence[j] == 'M'){  
   // 执行malloc操作  
   int size = rand() % (1024 \* 256) + 256;  
   int\* ptr = (int\*)malloc(size \* sizeof(int));  
   push(&stack, statistics\_index);  
   }else{  
   // 执行free操作  
   int index = pop(&stack);  
   free(statistics[index].ptr);  
   }  
   }  
  }

1. **结果统计**：

* printf("average\_malloc\_cost: %f ns\n", ((double)total\_malloc\_cost / (catalan\_number\*MALLOC\_NUM)));  
  printf("average\_free\_cost: %f ns\n", ((double)total\_free\_cost / (catalan\_number\*MALLOC\_NUM)));  
  printf("internal\_fragmentation\_rate: %f %%\n",   
   ((double)(total\_memory\_malloc\_size - total\_memory\_allocate\_size) / total\_memory\_malloc\_size)\*100);  
  printf("external\_fragmentation\_rate: %f %%\n",   
   ((double)(total\_external\_fragmentation\_rate / (catalan\_number\*MALLOC\_NUM)))\*100);

## 关键技术要点

### 1. 序列完整性保证

* 使用Catalan数确保生成的所有序列都是有效的malloc/free配对
* 通过递归算法生成所有可能的操作序列，保证测试的全面性

### 2. 内存信息精确获取

* 使用 mem\_getinfo() 获取实时内存使用情况
* 使用 mem\_findmax() 查询最大可用内存块大小
* 通过前后对比计算精确的内存分配大小

### 3. 时间测量精度

* 使用 sys\_timestamp() 获取高精度时间戳
* 在钩子函数中记录分配前后的时间差
* 所有时间统计以纳秒为单位

### 4. 资源管理

* 使用栈结构确保正确的内存释放顺序
* 动态分配序列存储空间
* 及时释放所有分配的内存避免内存泄漏

### 5. 随机化测试

* 随机生成分配大小(256字节到256KB)
* 通过不同的分配模式测试内存管理器的鲁棒性

## 测试指标解释

### 1. 内部碎片率

计算公式：(申请大小 - 实际分配大小) / 申请大小 × 100%

反映内存分配器因对齐或管理开销导致的空间浪费。

### 2. 外部碎片率

计算公式：最大可用块大小 / 总堆内存大小 × 100%

反映内存空间的连续性，值越小表示碎片化越严重。

### 3. 平均分配/释放时间

反映内存分配器的性能效率，包括查找合适内存块和维护内存结构的开销。

## 使用方法

1. **包含头文件**：确保包含必要的内存管理头文件
2. **调用测试函数**：直接调用 test\_memory\_allocate() 开始测试
3. **分析结果**：根据输出的统计数据分析内存管理性能
4. **查看内存状态**：测试结束后会调用 mem\_show() 显示最终内存状态

此测试模块可以有效评估内存管理系统的性能和碎片化情况，为内存管理器的优化提供重要的数据支持。通过全面的序列测试，能够发现各种边界情况下的性能问题。