# 线程级内存信息获取测试模块

## 概述

本模块实现了一个线程级内存使用信息获取和统计的测试系统。测试通过创建独立的工作线程，在线程内执行内存分配操作，使用 pthread\_get\_meminfo() 函数获取线程级别的内存使用统计信息，验证系统对单个线程内存使用情况的监控能力。

## 核心功能

### 1. 线程级内存统计

* **统计对象**：单个线程的内存分配和释放行为
* **统计指标**：累计分配大小、分配次数、累计释放大小、释放次数

### 2. 内存使用行为验证

* **验证机制**：通过分配前后的统计数据对比验证记录准确性
* **实时监控**：在内存操作执行的不同阶段获取统计信息

## 设计思路

### 1. 线程级内存监控

不同于系统级的全局内存统计，本模块专注于单个线程的内存使用行为分析：

struct pthread\_meminfo {  
 u32 sigma\_alloc\_size; // 累计分配大小  
 u32 alloc\_cnt; // 分配次数  
 u32 sigma\_free\_size; // 累计释放大小   
 u32 free\_cnt; // 释放次数  
};

通过 pthread\_get\_meminfo() 函数获取指定线程的内存使用统计，可以精确追踪每个线程的内存分配模式。

### 2. 测试验证流程

测试采用简单直接的验证方式：

1. **初始化**：创建线程并初始化内存统计结构
2. **分配操作**：在线程内执行大块内存分配
3. **统计获取**：分配前后分别获取内存统计信息
4. **数据对比**：验证统计数据的准确性

### 3. 内存操作模拟

int size = 8000;  
char\* string = (char\*)malloc(size \* sizeof(char));  
for(int i = 0; i < size; i++){  
 string[i] = 'a';  
}

分配8KB内存并进行写入操作，模拟真实的内存使用场景，确保分配的内存确实被使用。

## 配置参数

#define HIGH\_PRIORITY 120 // 测试线程优先级

## 代码结构

### 主要数据结构

// 线程内存信息结构（系统定义）  
struct pthread\_meminfo {  
 u32 sigma\_alloc\_size; // 累计分配的内存大小（字节）  
 u32 alloc\_cnt; // 内存分配操作次数  
 u32 sigma\_free\_size; // 累计释放的内存大小（字节）  
 u32 free\_cnt; // 内存释放操作次数  
};

### 核心函数

#### 1. task(void\* arg)

**功能**：测试线程的主要工作函数

**参数**：

* arg: 指向 pthread\_meminfo 结构的指针，用于存储内存统计信息

**实现逻辑**：

void\* task(void\* arg){  
 struct pthread\_meminfo\* task\_meminfo = (struct pthread\_meminfo\*)arg;  
   
 // 1. 执行内存分配操作  
 int size = 8000;  
 char\* string = (char\*)malloc(size \* sizeof(char));  
   
 // 2. 初始化分配的内存  
 for(int i = 0; i < size; i++){  
 string[i] = 'a';  
 }  
   
 // 3. 获取当前线程句柄  
 pthread\_t self = pthread\_self();  
   
 // 4. 输出分配前的内存统计信息  
 printk("before write: sigma\_alloc\_size: %u, alloc\_cnt: %u, sigma\_free\_size: %u, free\_cnt: %u\n",   
 task\_meminfo->sigma\_alloc\_size, task\_meminfo->alloc\_cnt,   
 task\_meminfo->sigma\_free\_size, task\_meminfo->free\_cnt);  
   
 // 5. 获取分配后的内存统计信息  
 ret = pthread\_get\_meminfo(self, task\_meminfo);  
 printk("after write: sigma\_alloc\_size: %u, alloc\_cnt: %u, sigma\_free\_size: %u, free\_cnt: %u\n",   
 task\_meminfo->sigma\_alloc\_size, task\_meminfo->alloc\_cnt,   
 task\_meminfo->sigma\_free\_size, task\_meminfo->free\_cnt);  
   
 // 6. 释放分配的内存  
 free(string);  
 return NULL;  
}

#### 2. test\_pthread\_get\_meminfo()

**功能**：主测试函数，协调整个测试流程

**执行流程**：

1. **初始化阶段**：

* pthread\_t task\_thread;  
  struct pthread\_meminfo task\_meminfo;  
  memset(&task\_meminfo, 0, sizeof(struct pthread\_meminfo));

1. **线程创建**：

* ret = pthread\_create2(&task\_thread, "task\_thread", HIGH\_PRIORITY, 0,   
   32 << 10, task, &task\_meminfo);  
  if(ret != 0){  
   printk("pthread\_create2 failed\n");  
   return;  
  }

1. **线程同步**：

* pthread\_join(task\_thread, NULL);

1. **结果输出**：

* printk("task memory info:\n");  
  printk("sigma\_alloc\_size: %u, alloc\_cnt: %u, sigma\_free\_size: %u, free\_cnt: %u\n",   
   task\_meminfo.sigma\_alloc\_size, task\_meminfo.alloc\_cnt,   
   task\_meminfo.sigma\_free\_size, task\_meminfo.free\_cnt);

1. **系统状态显示**：

* printk("final memory status:\n");  
  mem\_show();

## 关键技术要点

### 1. 线程级内存追踪

* 使用 pthread\_get\_meminfo() 系统调用获取特定线程的内存使用统计
* 通过 pthread\_self() 获取当前线程句柄进行自我查询
* 支持跨线程的内存信息查询（主线程查询工作线程）

### 2. 内存操作验证

* 不仅分配内存，还进行实际的写入操作确保内存被真正使用
* 通过字符填充验证内存分配的有效性
* 完整的分配-使用-释放流程测试

### 3. 统计数据完整性

* 记录分配前后的完整统计信息
* 包含操作次数和累计大小两个维度的统计
* 支持增量统计分析（通过前后对比）

### 4. 线程安全设计

* 每个线程维护独立的内存统计信息
* 避免多线程环境下的统计数据竞争
* 线程句柄作为查询的唯一标识

## 测试指标解释

### 1. sigma\_alloc\_size（累计分配大小）

记录线程从创建到查询时刻累计分配的内存总量，单位为字节。

### 2. alloc\_cnt（分配次数）

记录线程执行 malloc() 或相关内存分配函数的总次数。

### 3. sigma\_free\_size（累计释放大小）

记录线程累计释放的内存总量，单位为字节。

### 4. free\_cnt（释放次数）

记录线程执行 free() 或相关内存释放函数的总次数。

## 预期输出示例

在调用 ret = pthread\_get\_meminfo(self, task\_meminfo);函数之后，可以准确获取本任务的堆内存使用情况。包括了 sigme\_alloc\_size, alloc\_cnt, sigma\_free\_size, free\_cnt。在执行时候，应当准确的打印出这些字段。但是在实际测试的时候，这四个字段的值都是一致的，初步推测为直接保存了物理地址，发生了UB行为。如图 test\_pthread\_get\_meminfo所示