# 任务抢占和优先级倒置测试模块

## 概述

本模块实现了一个综合的任务抢占和优先级倒置测试系统，用于评估实时操作系统的任务调度性能。测试通过创建不同优先级的任务，模拟资源竞争场景，收集和分析优先级倒置和抢占响应时间的统计数据。

## 核心功能

### 1. 优先级反转检测

* **检测机制**：监控高优先级任务因低优先级任务占用资源而被阻塞的情况
* **阈值设定**：当阻塞时间超过1.5ms时记录为可能发生优先级反转，同时打印任务开始执行序列来检查是否发生真正发生了优先级反转

### 2. 抢占响应时间测量

* **测量对象**：从任务就绪到实际开始执行的时间延迟
* **性能指标**：平均响应时间、最大/最小响应时间
* **慢抢占检测**：响应时间超过3ms的抢占被标记为慢抢占

## 设计思路

### 1.优先级反转的检测

优先级反转是指：低优先级的任务占有资源，此时系统中加入众多的中优先级且资源无关的任务，由于抢占型调度，低优先级任务被挂起。此时又加入高优先级且需要获取资源的任务，由于低优先级任务被挂起且占有资源（资源无法释放），导致高优先级任务被迫阻塞。因此本模块中采用 ret = sem\_wait(task\_semid);，即信号量的获取时间来推测是否发生了优先级反转。如果被阻塞时间过长，标记为可能发生了优先级反转，此时会打印出任务的执行序列来辅助判断是否发生了真正的优先级反转。

### 2.抢占响应时间的测量

抢占响应时间是指较高优先级任务从被创建到真正执行所用的时间。在本模块中，通过判断是否传入了任务创建的时间戳来区分高优先级任务与低优先级任务。

typedef struct {  
 u64 ready\_time; // Task ready time  
} task\_args\_t;

只要在高优先级任务的执行开始，记录任务的执行时间，在和传入的参数中的任务创建的时间戳相减，即可获得任务对应的抢占响应时间。同时，由于本模块是多线程的，并且在执行中要更新最大抢占时间，存在着数据同步风险，因此本模块中存在数据同步风险的部分统一采用了GCC所提供的 \_\_sync\_\* 原子操作函数来确保数据同步。

// Calculate preemption response time  
u64 preempt\_response\_ns = (exec\_start - args->ready\_time) \* 1000000000 / freq;  
// Update preemption response time statistics  
\_\_sync\_fetch\_and\_add(&stats.total\_preemptions, 1);  
\_\_sync\_add\_and\_fetch(&stats.total\_preempt\_ns, preempt\_response\_ns);

### 3.测试思路

在代码中，主要是由run\_one\_round() 来执行每一轮的测试，作用是在一轮中**随机生成若干高、中、低优先级线程**，模拟典型的优先级反转场景。高优先级线程竞争资源，低优先级线程先获取资源，中优先级线程则不参与资源竞争但可能频繁调度，从而打断低优先级线程的执行。

线程在创建后先挂起，打乱恢复顺序后再依次恢复，模拟真实调度中的不确定性。线程执行过程中会统计是否因信号量等待而发生优先级反转，并记录抢占响应时间等关键指标。如果发生了信号量等待时间过长，那么会打印任务的启动顺序来最终确定是否发生了优先级反转。

每轮结束后回收资源，并在检测到反转时输出相关信息，为最终统计提供数据支撑。该函数作为单轮测试单位，为整个测试提供实验基础。

## 配置参数

#define NUM\_ROUNDS 2000 // 测试轮数  
#define STACK\_SIZE (32 << 10) // 任务栈大小  
#define HIGH\_PRIORITY\_BASE 120 // 高优先级基值  
#define MID\_PRIORITY\_BASE 100 // 中优先级基值  
#define LOW\_PRIORITY\_BASE 80 // 低优先级基值  
#define MAX\_TASK\_NUM 15 // 最大任务数

## 代码结构

### 主要数据结构

// 统计数据结构  
typedef struct {  
 int total\_inversions; // 总优先级倒置次数  
 u64 total\_delay\_ns; // 总阻塞时间（纳秒）  
 u64 max\_delay\_ns; // 最大阻塞时间  
 int severe\_inversions; // 严重倒置次数（>4ms）  
 int total\_preemptions; // 总抢占次数  
 u64 total\_preempt\_ns; // 总抢占响应时间  
 u64 max\_preempt\_ns; // 最大抢占响应时间  
 u64 min\_preempt\_ns; // 最小抢占响应时间  
 int slow\_preemptions; // 慢抢占次数（>3ms）  
} statistics\_t;

### 核心函数

#### 1. mutex\_task(void\* arg)

**功能**：竞争资源的任务函数  
**参数**：

* arg: 指向 task\_args\_t 结构的指针，包含任务就绪时间

**实现逻辑**：

void\* mutex\_task(void\* arg) {  
 // 1. 计算抢占响应时间  
 u64 exec\_start = sys\_timestamp();  
 if(args != NULL) {  
 u64 preempt\_response\_ns = (exec\_start - args->ready\_time) \* 1000000000 / freq;  
 // 更新抢占响应时间统计  
 }  
 // 2. 尝试获取信号量（模拟资源竞争）  
 u64 lock\_start = sys\_timestamp();  
 sem\_wait(task\_semid);  
 u64 lock\_end = sys\_timestamp();  
 // 3. 计算阻塞时间并检测优先级倒置  
 u64 delay\_ns = (lock\_end - lock\_start) \* 1000000000 / freq;  
 if (delay\_ns > 1500) { // 超过1.5ms视为倒置  
 // 更新倒置统计  
 }  
 // 4. 模拟工作负载  
 udelay(delay\_time);  
 // 5. 释放信号量  
 sem\_post(task\_semid);  
}

#### 2. no\_mutex\_task(void\* arg)

**功能**：不竞争资源的任务函数  
**作用**：模拟系统中的其他任务，用于测试任务切换开销

#### 3. run\_one\_round()

**功能**：执行一轮测试的核心函数

**测试流程**：

1. **随机生成任务数量**：
   * 总任务数：1-TASK\_NUM个
   * 高优先级任务数：随机分配
   * 低优先级任务数：随机分配
   * 非竞争任务数：剩余任务
2. **创建任务**：

* // 创建高优先级竞争任务  
  pthread\_create2(&high\_priority\_mutex\_tid[i], "high\_priority\_mutex\_tid",   
   HIGH\_PRIORITY\_BASE, 0, STACK\_SIZE, mutex\_task, &high\_args[i]);  
    
  // 创建低优先级竞争任务  
  pthread\_create2(&low\_priority\_mutex\_tid[i], "low\_priority\_mutex\_tid",   
   LOW\_PRIORITY\_BASE, 0, STACK\_SIZE, mutex\_task, NULL);  
    
  // 创建中优先级非竞争任务  
  pthread\_create2(&no\_mutex\_tid[i], "no\_mutex\_tid",   
   MID\_PRIORITY\_BASE, 0, STACK\_SIZE, no\_mutex\_task, NULL);

1. **任务调度**：
   * 所有任务创建后先挂起
   * 使用随机顺序恢复任务执行
   * 通过 pthread\_join() 等待所有任务完成

#### 4. test\_task\_preempt()

**功能**：主测试函数

**执行流程**：

1. **初始化**：

* srand(time(NULL)); // 随机数种子  
  memset(&stats, 0, sizeof(stats)); // 清零统计数据  
  stats.min\_preempt\_ns = UINT64\_MAX; // 初始化最小值

1. **多轮测试**：
   * 执行2000轮测试
   * 每轮使用不同的信号量实例
   * 每100轮输出进度信息
2. **结果输出**：

* printk("=========== Priority Inversion and Preemption Response Test Report ===========\n");  
  printk("Test completion: %d/%d rounds successful\n", success\_rounds, NUM\_ROUNDS);  
    
  // 优先级倒置统计  
  printk("Total priority inversions: %d\n", stats.total\_inversions);  
  printk("Average blocking time: %10d ns\n", stats.total\_delay\_ns / stats.total\_inversions / 1000);  
    
  // 抢占响应时间统计  
  printk("Total preemptions: %d\n", stats.total\_preemptions);  
  printk("Average preemption response time: %10d ns\n", stats.total\_preempt\_ns / stats.total\_preemptions / 1000);

## 关键技术要点

### 1. 线程安全的统计更新

使用原子操作确保多线程环境下统计数据的准确性：

\_\_sync\_fetch\_and\_add(&stats.total\_inversions, 1);  
\_\_sync\_add\_and\_fetch(&stats.total\_delay\_ns, delay\_ns);  
\_\_sync\_bool\_compare\_and\_swap(&stats.max\_delay\_ns, current\_max, delay\_ns);

### 2. 时间测量精度

* 使用 sys\_timestamp() 获取高精度时间戳
* 通过 get\_cntfrq() 获取系统频率进行时间转换
* 所有时间统计以纳秒为单位

### 3. 资源管理

* 动态分配任务数组和参数数组
* 及时释放内存避免内存泄漏
* 正确关闭和取消链接信号量

### 4. 错误处理

* 检查所有系统调用的返回值
* 在错误情况下正确清理资源
* 提供详细的错误信息输出

## 使用方法

1. **包含头文件**：确保包含必要的系统头文件
2. **调用测试函数**：直接调用 test\_task\_preempt() 开始测试
3. **分析结果**：根据输出的统计报告分析系统性能

此测试模块可以有效评估实时系统的调度性能，识别潜在的优先级倒置问题，为系统优化提供数据支持。