基于抽象代数的 GroupLock 锁机制

摘要

本项目将抽象代数中的群论应用于 xv6 操作系统，设计并实现了一种新型的同步原语——GroupLock。通过将锁状态建模为有限群 Z_2 的元素，利用群运算的数学性质实现互斥访问，为操作系统同步机制提供了严格的数学理论基础和可形式化证明的正确性保证。

关键词: 抽象代数, 群论, 操作系统, 同步原语, 形式化验证

1. 引言

1.1 研究背景

传统的操作系统锁机制主要基于硬件原子指令和工程经验，缺乏严格的数学理论支撑。本研究探索将抽象代数理论应用于系统编程，为同步机制提供数学基础和形式化证明。

1.2 研究目标

- 设计基于群论的锁机制
- 提供可形式化证明的正确性保证
- 实现高效简单的同步原语
- 提供一种跨学科的研究示例
- 从不同的角度去理解和实现操作系统同步机制
- 展示数学理论在系统编程中的应用价值

2. 数学理论基础

2.1 群论基础

定义: 群是一个代数结构 $G = (S, \bigcirc)$，其中 S 是非空集合，$\bigcirc$ 是二元运算，满足：

1. **封闭性**: $\forall a, b \in S, a \bigcirc b \in S$
2. **结合律**: $\forall a, b, c \in S, (a \bigcirc b) \bigcirc c = a \bigcirc (b \bigcirc c)$
3. **单位元**: $\exists e \in S, \forall a \in S, e \bigcirc a = a \bigcirc e = a$
4. **逆元**: $\forall a \in S, \exists a^{-1} \in S, a \bigcirc a^{-1} = a^{-1} \bigcirc a = e$

2.2 Z_2 群的选择

本研究选择二元群 $Z_2 = (\{0, 1\}, +)$ 作为锁状态空间：

- **群元素**: $\{0, 1\}$
- **群运算**: 模 2 加法 $(+)$
- **单位元**: 0 (解锁状态)
- **逆元**: 每个元素是自己的逆元

运算表:

$+$	**0**	**1**
0	0	1
1	1	0

2.3 锁机制的群论建模

状态映射:

- $0 \rightarrow$ UNLOCKED (解锁状态)
- $1 \rightarrow$ LOCKED (加锁状态)

操作映射:

- $\text{acquire}() \rightarrow \text{state} + 1 \pmod 2$
- $\text{release}() \rightarrow \text{state} + 1 \pmod 2$

3. 完整代码实现

> ⓘ **Info:** 具体代码请参考各自文件

3.1 头文件定义

kernel/grouplock.h

```
#ifndef GROUPLOCK_H
#define GROUPLOCK_H

#include "types.h"
#include "param.h"
#include "spinlock.h"

// Maximum number of group locks
#define MAX_GROUPLOCKS 64

// Z/2Z group element type
typedef enum {
    GROUP_ELEM_0 = 0,    // Unlocked state, identity element
    GROUP_ELEM_1 = 1     // Locked state
} group_element_t;

// Group lock structure
struct grouplock {
    volatile group_element_t state;  // Current group element state
    int group_id;                    // Group lock ID
    int holder_pid;                  // Process ID holding the lock
    char name[16];                   // Lock name
    int ref_count;                   // Reference count
    uint64 acquire_time;             // Lock acquisition timestamp
    struct spinlock debug_lock;      // Lock protecting debug information
};

// Group operation functions
group_element_t group_add(group_element_t a, group_element_t b);
group_element_t group_inverse(group_element_t a);
```

```c
int group_is_identity(group_element_t a);

// Group lock operation functions
void grouplock_init(void);
int grouplock_create(int group_id, char *name);
int grouplock_acquire(int group_id);
int grouplock_release(int group_id);
int grouplock_destroy(int group_id);
void grouplock_debug_info(int group_id);

// Mathematical verification functions
int verify_group_properties(void);
int verify_deadlock_freedom(void);
int verify_atomic_group_operations(void);

#endif
```

3.2 核心实现

kernel/grouplock.c

```
// Global group lock table
static struct grouplock grouplocks[MAX_GROUPLOCKS];
static struct spinlock grouplocks_table_lock;  // It ensures that one process
doesn't try to destroy a locke
                                               // while another is trying to check
if it exists.

// Next, we will only show the two core functions

// === Core lock operation: group theory based acquire ===

int grouplock_acquire(int group_id) {
    if (group_id < 0 || group_id >= MAX_GROUPLOCKS) {
        return -1;
    }

    struct proc *p = myproc();

    // Check if lock exists(check whether lock has been created or not)
    acquire(&grouplocks_table_lock);
    if (grouplocks[group_id].group_id == -1) {
        release(&grouplocks_table_lock);
        return -2;
    }
    release(&grouplocks_table_lock);

    // Disable interrupts to avoid deadlock
    push_off();

    printf("GroupLock: Process %d attempting to acquire lock %d\n", p->pid,
group_id);

    // Use atomic CAS for group operation: can acquire lock only when current
state is identity
    while (1) {
```

```
    group_element_t expected = GROUP_ELEM_0;  // Expect unlocked state (identity)
    group_element_t desired = GROUP_ELEM_1;   // Want to set to locked state

    // Atomic compare-and-swap: atomic implementation of group operation 0 + 1 = 1
        if (__sync_bool_compare_and_swap(&grouplocks[group_id].state, expected,
desired)) {
            // Successfully acquired lock: applied group operation e + a = a
            grouplocks[group_id].holder_pid = p->pid;
            grouplocks[group_id].acquire_time = ticks;

            // Memory barrier ensures critical section operations are not
reordered before lock acquisition
            __sync_synchronize();

            printf("GroupLock: Process %d acquired lock %d using group operation
(0 + 1 = 1)\n",
                   p->pid, group_id);

            pop_off();
            return 0;
        }

    // If acquisition fails, yield CPU (spin wait)
        pop_off();
        yield();
        push_off(); // To ensure next iteration has interrupts off
    }
}

// === Core lock operation: group theory based release ===

int grouplock_release(int group_id) {
    if (group_id < 0 || group_id >= MAX_GROUPLOCKS) {
        return -1;
    }

    struct proc *p = myproc();
```

```
    // Check if lock exists(check whether lock has been created or not)
    acquire(&grouplocks_table_lock);
    if (grouplocks[group_id].group_id == -1) {
        release(&grouplocks_table_lock);
        return -2;
    }
    release(&grouplocks_table_lock);

    // Verify if current process is lock holder
    if (grouplocks[group_id].holder_pid != p->pid) {
        return -3;
    }

    push_off();

    // Clear holder information
    grouplocks[group_id].holder_pid = -1;
    grouplocks[group_id].acquire_time = 0;

    // Memory barrier ensures critical section operations are completed before releasing lock
    __sync_synchronize();

    printf("GroupLock: Process %d releasing lock %d using inverse operation\n", p->pid, group_id);

    // Atomically apply group inverse operation: 1 + 1 = 0 (mod 2)
    group_element_t old_state = atomic_group_add(&grouplocks[group_id].state, GROUP_ELEM_1);

    if (old_state != GROUP_ELEM_1) {
    printf("GroupLock: WARNING - Released lock from unexpected state %d\n", old_state);
    } else {
        printf("GroupLock: Process %d released lock %d using group operation (1 + 1 = 0)\n",
               p->pid, group_id);
    }
```

```
    pop_off();
    return 0;
}

//...
```

3.3 系统调用集成

3.3.1 系统调用表声明

kernel/syscall.c

```
extern uint64 sys_grouplock_create(void);
extern uint64 sys_grouplock_acquire(void);
extern uint64 sys_grouplock_release(void);
extern uint64 sys_grouplock_verify(void);
extern uint64 sys_grouplock_destroy(void);
extern uint64 sys_grouplock_debug(void);
```

3.3.2 系统调用实现

kernel/sysproc.c

```
  uint64 sys_grouplock_create(void) {
    int group_id;
    char name[16];

    argint(0, &group_id);
    if (argstr(1, name, 16) < 0) {
        return -1;
    }

    return grouplock_create(group_id, name);
}

uint64 sys_grouplock_acquire(void) {
    int group_id;

    argint(0, &group_id);

    return grouplock_acquire(group_id);
}

uint64 sys_grouplock_release(void) {
    int group_id;

    argint(0, &group_id);
```

```
    return grouplock_release(group_id);
}

uint64 sys_grouplock_destroy(void) {
    int group_id;

    argint(0, &group_id);

    return grouplock_destroy(group_id);
}

uint64 sys_grouplock_verify(void) {
    int result1 = verify_group_properties();
    int result2 = verify_deadlock_freedom();
    int result3 = verify_atomic_group_operations();

    return (result1 == 0 && result2 == 0 && result3 == 0) ? 0 : -1;
}

uint64 sys_grouplock_debug(void) {
    int group_id;

    argint(0, &group_id);

    grouplock_debug_info(group_id);
    return 0;
}
```

3.4 用户态接口

user/user.h

```
int grouplock_create(int group_id, char *name);
int grouplock_acquire(int group_id);
int grouplock_release(int group_id);
int grouplock_destroy(int group_id);
int grouplock_verify(void);
int grouplock_debug(int group_id);
```

3.5 测试程序及结果

3.5.1 基础操作和性质测试程序

user/grouplocktest.c

```
// By code to verify the group theory to make sure
// there is no problem with the underlying principle
void test_mathematical_properties(void) {
    printf("\n=== Mathematical Properties Verification Test ===\n");

    int result = grouplock_verify();
    TEST_ASSERT(result == 0, "Group theory mathematical properties verification
passed");

    if (result == 0) {
        printf("Verification content:\n");
        printf("  - Z/2Z group closure property ✓\n");
        printf("  - Associativity ✓\n");
        printf("  - Commutativity (Abelian group property) ✓\n");
        printf("  - Identity element existence ✓\n");
        printf("  - Inverse element existence ✓\n");
        printf("  - Deadlock freedom mathematical proof ✓\n");
        printf("  - Atomic group operation verification ✓\n");
    }
}

void test_group_theory_properties(void) {
    printf("=== Group Theory Properties Practical Verification ===\n");

    if (grouplock_create(6, "theory_lock") < 0) {
        printf("✗ Failed to create theory test lock\n");
        tests_failed++;
        return;
    }

    printf("Verifying specific applications of group operations:\n");

    // Verify identity property: e + a = a
    printf("1. Identity property: Initial state is 0 (identity element)\n");
```

```
    grouplock_debug(6);

    // Verify group operation: 0 + 1 = 1
    printf("2. Group operation: 0 + 1 = 1 (acquire operation)\n");
    if (grouplock_acquire(6) == 0) {
        grouplock_debug(6);

        // Verify inverse operation: 1 + 1 = 0
        printf("3. Inverse operation: 1 + 1 = 0 (release operation)\n");
        grouplock_release(6);
        grouplock_debug(6);

        TEST_ASSERT(1, "Group theory properties verified in practical
operations");
    }

    grouplock_destroy(6);
}

// Test the basic operations of the group lock, like create, acquire, release,
destroy
void test_basic_operations(void) {
    printf("\n=== Basic Operations Test ===\n");

    // Create group lock
    int result = grouplock_create(1, "test_lock");
    TEST_ASSERT(result == 0, "Successfully created group lock 1");

    // Acquire lock (0 + 1 = 1)
    result = grouplock_acquire(1);
    TEST_ASSERT(result == 0, "Successfully acquired group lock 1 (Group operation:
0 + 1 = 1)");

    // Release lock (1 + 1 = 0)
    result = grouplock_release(1);
    TEST_ASSERT(result == 0, "Successfully released group lock 1 (Group operation:
1 + 1 = 0)");
```

```
    // Acquire and release again to verify repeatability
    result = grouplock_acquire(1);
    TEST_ASSERT(result == 0, "Can repeatedly acquire group lock 1");

    result = grouplock_release(1);
    TEST_ASSERT(result == 0, "Can repeatedly release group lock 1");

    // Destroy lock
    result = grouplock_destroy(1);
    TEST_ASSERT(result == 0, "Successfully destroyed group lock 1");
}
```

3.5.2 性能基准测试

user/grouplock_benchmark.c

```
// Test whether there will be problems in acquiring the same grouplock in multiple concurrent situations
test_concurrent_access();

// Test multiple processes competing for the same lock
// to verify the performance of locks under high concurrency
// and verify the correctness and fairness of locks
test_multiple_processes();

//Test some cases like invalid ID, repeated operations, destroying a lock in use
test_edge_cases();

// Test lock contention with multiple processes incrementing a shared counter in a file
test_lock_contention();
```

3.5.3 测试结果

```
=== Test Results Summary ===
Tests passed: 16
Tests failed: 0
Total tests: 16
All tests passed! GroupLock mechanism works correctly.
Mathematical theory and system implementation perfectly combined!
```

4. 数学理论证明

4.1 互斥性证明

定理 1 (互斥性) GroupLock 机制保证互斥访问

证明: 设有进程 P_1, P_2 同时尝试获取锁 L。

1. **初始状态**: L.state = 0 (单位元)
2. **原子性保证**: 使用 CAS 指令确保状态检查和修改的原子性
3. **互斥条件**: 只有当 L.state = 0 时，CAS(L.state, 0, 1) 才能成功
4. **唯一成功**: 由于 CAS 的原子性，最多只有一个进程能成功执行状态转换
5. **群运算语义**: 成功的进程执行了群运算 $0 + 1 = 1$

因此，在任意时刻，最多只有一个进程能持有锁。

4.2 死锁自由性证明

定理 2 (死锁自由性) GroupLock 机制保证死锁自由性

证明: 基于 Z_2 群的数学性质：

1. **有限状态空间**: $S = \{0, 1\}$，状态空间有限
2. **可达性**: $\forall s \in S, \exists n \in \mathbb{N}, s + n \cdot 1 = 0 (\mathrm{mod}\, 2)$
    - 具体地：$0 + 0 \cdot 1 = 0, 1 + 1 \cdot 1 = 0$
3. **无循环等待**: 每个状态都可以在有限步内到达单位元
4. **逆元保证**: 每个元素都有逆元，确保可以“撤销”操作

因此，系统不存在永久阻塞状态。

4.3 公平性分析

定理 3 (公平性) GroupLock 具有数学保证的公平性基础

证明: 基于 Z_2 群的交换律：

1. **交换律**: $\forall a, b \in Z_2, a + b = b + a$
2. **操作等价性**: 任何操作序列的最终结果与操作顺序无关
3. **调度无关性**: 底层调度器的公平性决定了上层锁的公平性

群的数学性质为公平调度提供了理论基础.

5. 性能分析与比较

5.1 理论复杂度

操作	时间复杂度	空间复杂度	数学操作
acquire	$O(1)$	$O(1)$	群运算 $0+1$
release	$O(1)$	$O(1)$	群运算 $1+1$
verify	$O(1)$	$O(1)$	群性质检查

5.2 与传统锁的比较

特性	xv6 Spinlock	GroupLock	优势
理论基础	硬件原子指令	抽象代数群论	数学严谨性
可证明性	经验性	数学证明	形式化保证
扩展性	有限	群结构可扩展	理论指导
教育价值	工程实践	理论与实践结合	跨学科应用

6. 扩展应用

6.1 其他群结构的猜想应用

1. **循环群** Z_n: 可实现 n 级锁状态
2. **置换群**: 可实现复杂的锁排序
3. **非阿贝尔群**: 处理非交换的同步模式

6.2 读写锁的猜想代数结构

可以通过使用格结构来表示读写锁

- 通过 `Join` 和 `Meet` 操作来表示状态转换
- 通过 `Bottom` 和 `Top` 表示边界或冲突情况

7. 结论

7.1 主要贡献

1. **理论创新**: 首次将抽象代数系统地应用于操作系统同步原语
2. **数学严谨性**: 提供可证明的正确性保证
3. **实用性**: 在 xv6 中成功实现并验证
4. **教育价值**: 展示数学理论在系统编程中的应用

7.2 技术特点

- **形式化建模**: 使用群论对锁状态和操作进行数学建模
- **原子群运算**: 通过原子 CAS 指令实现群运算的原子性
- **数学验证**: 提供完整的群性质验证和正确性证明
- **高性能**: $O(1)$时间复杂度，与传统锁相当

7.3 未来工作

1. **扩展到其他群结构**: 探索更复杂的同步模式
2. **形式化验证**: 使用定理证明器验证实现的正确性
3. **性能优化**: 基于群性质的调度算法优化
4. **应用推广**: 在更多操作系统中实现和测试

8. 参考文献

1. Abstract Algebra: Theory and Applications
2. xv6: a simple, Unix-like teaching operating system, MIT PDOS
3. Operating System Concepts, Abraham Silberschatz
4. Operating Systems: Three Easy Pieces

致谢: 感谢 MIT xv6 项目提供的优秀教学平台，使得理论与实践的结合成为可能。

项目地址: https://github.com/ice345/xv6-riscv-src_development

作者: Junbin Liang