GroupLock: A Synchronization Mechanism Based on Abstract Algebra

Abstract

This project applies group theory from abstract algebra to the xv6 operating system to design and implement a novel synchronization primitive—GroupLock. By modeling the lock state as elements of the finite group Z_2, we leverage the mathematical properties of group operations to achieve mutual exclusion. This provides a rigorous mathematical foundation and a formally verifiable correctness guarantee for operating system synchronization mechanisms.

Keywords: Abstract Algebra, Group Theory, Operating Systems, Synchronization Primitives, Formal Verification

1. Introduction

1.1 Research Background

Traditional operating system lock mechanisms are primarily based on hardware atomic instructions and engineering experience, often lacking rigorous mathematical-theoretical support. This research explores the application of abstract algebra theory to systems programming, aiming to provide a mathematical foundation and formal proof for synchronization mechanisms.

1.2 Research Objectives

- Design a lock mechanism based on group theory.
- Provide a formally verifiable correctness guarantee.
- Implement an efficient and simple synchronization primitive.
- Offer an example of interdisciplinary research.
- Understand and implement OS synchronization from a different perspective.
- Demonstrate the value of mathematical theory in systems programming.

2. Mathematical Foundations

2.1 Basics of Group Theory

Definition: A group is an algebraic structure $G = (S, \bigcirc)$, where S is a non-empty set and $\bigcirc$ is a binary operation, satisfying:

1. **Closure**: $\forall a, b \in S, a \bigcirc b \in S$
2. **Associativity**: $\forall a, b, c \in S, (a \bigcirc b) \bigcirc c = a \bigcirc (b \bigcirc c)$
3. **Identity Element**: $\exists e \in S, \forall a \in S, e \bigcirc a = a \bigcirc e = a$
4. **Inverse Element**: $\forall a \in S, \exists a^{-1} \in S, a \bigcirc a^{-1} = a^{-1} \bigcirc a = e$

2.2 Choice of the Z_2 Group

This research selects the binary group $Z_2 = (\{0, 1\}, +)$ as the lock state space:

- **Group Elements**: $\{0, 1\}$
- **Group Operation**: Addition modulo 2 (+)
- **Identity Element**: 0 (Unlocked state)
- **Inverse Element**: Every element is its own inverse.

Operation Table:

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

2.3 Group-Theoretic Modeling of the Lock Mechanism

State Mapping:

- 0 $\rightarrow$ UNLOCKED
- 1 $\rightarrow$ LOCKED

Operation Mapping:

- acquire() $\rightarrow$ state + 1 (mod 2)
- release() $\rightarrow$ state + 1 (mod 2)

3. Complete Code Implementation

ⓘ **Info:** For specific code, please refer to the respective files.

3.1 Header File Definition

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);
int group_is_identity(group_element_t a);

// Group lock operation functions
```

```c
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 Core Implementation

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 System Call Integration

3.3.1 System Call Table Declaration

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 System Call Implementation

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) {
```

```c
    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 User-space Interface

user/user.h

```c
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 Test Programs and result

3.5.1 Basic Operations and Properties Test

user/grouplocktest.c

```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 Performance Benchmark

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 Test result

```
=== 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. Mathematical Proofs

4.1 Proof of Mutual Exclusion

Theorem 1 (Mutual Exclusion) The GroupLock mechanism guarantees mutually exclusive access.

Proof: Let processes P_1 and P_2 attempt to acquire lock L simultaneously.

1. **Initial State**: L.state = 0 (identity element)
2. **Atomicity Guarantee**: The CAS instruction ensures atomicity of state checking and modification.
3. **Mutual Exclusion Condition**: CAS(L.state, 0, 1) can only succeed if L.state = 0.
4. **Unique Success**: Due to the atomicity of CAS, at most one process can successfully execute the state transition.
5. **Group Operation Semantics**: The successful process performs the group operation $0 + 1 = 1$.

Therefore, at any given moment, at most one process can hold the lock.

4.2 Proof of Deadlock Freedom

Theorem 2 (Deadlock Freedom) The GroupLock mechanism is deadlock-free.

Proof: Based on the mathematical properties of the Z_2 group:

1. **Finite State Space**: $S = \{0, 1\}$, the state space is finite.
2. **Reachability**: $\forall s \in S, \exists n \in \mathbb{N}, s + n \cdot 1 = 0 (\bmod 2)$
    - Specifically: $0 + 0 \cdot 1 = 0, 1 + 1 \cdot 1 = 0$
3. **No Circular Wait**: Every state can reach the identity element in a finite number of steps.
4. **Inverse Guarantee**: Every element has an inverse, ensuring that operations can be "undone".

Therefore, the system has no permanently blocked states.

4.3 Fairness Analysis

Theorem 3 (Fairness) GroupLock has a mathematically guaranteed basis for fairness.

Proof: Based on the commutative property of the Z_2 group:

1. **Commutativity**: $\forall a, b \in Z_2, a + b = b + a$
2. **Operational Equivalence**: The final result of any sequence of operations is independent of the order of operations.
3. **Scheduler Independence**: The fairness of the underlying scheduler determines the fairness of the lock at a higher level.

The mathematical properties of the group provide a theoretical basis for fair scheduling.

5. Performance Analysis and Comparison

5.1 Theoretical Complexity

Operation	Time Complexity	Space Complexity	Mathematical Operation
acquire	$O(1)$	$O(1)$	Group op $0 + 1$
release	$O(1)$	$O(1)$	Group op $1 + 1$
verify	$O(1)$	$O(1)$	Check group properties

5.2 Comparison with Traditional Locks

Feature	xv6 Spinlock	GroupLock	Advantage
Theoretical Basis	Hardware atomic instructions	Abstract algebra (Group Theory)	Mathematical rigor
Provability	Empirical	Mathematical proof	Formal guarantee
Extensibility	Limited	Group structure is extensible	Theory-guided
Educational Value	Engineering practice	Theory meets practice	Interdisciplinary application

6. Extended Applications

6.1 Conjectured Applications of Other Group Structures

1. **Cyclic Group Z_n**: Could implement n-level lock states.
2. **Permutation Group**: Could implement complex lock ordering.
3. **Non-Abelian Group**: Could handle non-commutative synchronization patterns.

6.2 Conjectured Algebraic Structure for Read-Write Locks

A lattice structure could be used to represent read-write locks:

- Use `Join` and `Meet` operations to represent state transitions.
- Use `Bottom` and `Top` to represent boundary or conflict conditions.

7. Conclusion

7.1 Main Contributions

1. **Theoretical Innovation**: First systematic application of abstract algebra to an OS synchronization primitive.
2. **Mathematical Rigor**: Provides a provable correctness guarantee.
3. **Practicality**: Successfully implemented and validated in xv6.
4. **Educational Value**: Demonstrates the application of mathematical theory in systems programming.

7.2 Technical Features

- **Formal Modeling**: Uses group theory for mathematical modeling of lock states and operations.
- **Atomic Group Operations**: Implements atomic group operations using atomic CAS instructions.
- **Mathematical Verification**: Provides complete verification of group properties and correctness proofs.
- **High Performance**: $O(1)$ time complexity, comparable to traditional locks.

7.3 Future Work

1. **Extend to Other Group Structures**: Explore more complex synchronization patterns.
2. **Formal Verification**: Use theorem provers to verify the implementation's correctness.
3. **Performance Optimization**: Optimize scheduling algorithms based on group properties.
4. **Broader Application**: Implement and test in more operating systems.

8. References

[1] T. W. Judson, *Abstract Algebra: Theory and Applications*. Annual Reviews, 2022.

[2] MIT PDOS, "xv6: A simple, Unix-like teaching operating system," Massachusetts Institute of Technology, 2019.

[3] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. John Wiley & Sons, 2018.

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.01 ed. Arpaci-Dusseau Books, 2018.

Acknowledgments: Thanks to the MIT xv6 project for providing an excellent teaching platform that made the combination of theory and practice possible.

Project Repository: https://github.com/ice345/xv6-riscv-src_development

Author: Junbin Liang