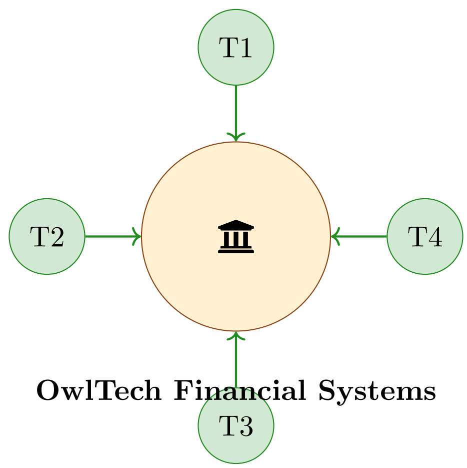
Project 1

Multi-Threaded Banking System

Four-Phase Threading Implementation



CS 3502: Operating Systems

Department of Computer Science
College of Computing and Software Engineering
Kennesaw State University

1 Introduction

OwlTech Financial Systems Division

Welcome to OwlTech Financial Systems! Your team has been tasked with developing a robust multi-threaded transaction processing system. This project will challenge you to implement proper thread synchronization, handle concurrent access to shared resources, and solve one of the classic problems in systems programming: deadlock.
You'll build this system in four phases, each adding complexity to demonstrate key threading concepts used in real-world financial systems.

This project focuses on multi-threading concepts essential to operating systems:

- Creating and managing multiple threads
- Protecting shared resources with synchronization primitives
- Understanding and creating deadlock conditions
- Implementing deadlock resolution strategies

Important: While we use a banking system as our example, you may implement these concepts using any scenario that properly demonstrates the required threading behaviors.

2 Project Overview

You will implement a multi-threaded application in four progressive phases:

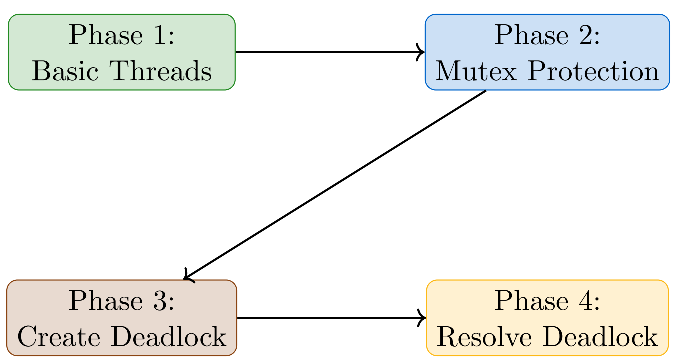


Figure 1: Project Phases

2.1 Banking System Example

The classic banking system demonstrates all required concepts:

- **Accounts**: Shared resources that multiple threads access
- **Transactions**: Operations performed by threads
- **Tellers/ATMs**: Threads processing transactions
- **Transfers**: Operations requiring multiple resources (deadlock potential)

3 Phase 1: Basic Thread Operations

Create a program demonstrating basic multi-threading without synchronization.

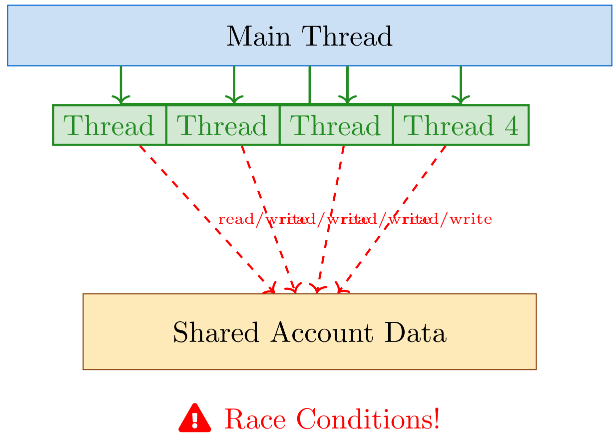


Figure 2: Phase 1: Multiple threads accessing shared data without protection

3.1 Requirements

- Create multiple threads (at least 2, but try for more to see interesting behaviors)
- Each thread should perform multiple operations
- Display thread IDs and operation details
- Show the race condition problem (incorrect results due to unsynchronized access)

What is a Race Condition?

A **race condition** occurs when multiple threads access shared data simultaneously without synchronization. The final result depends on the unpredictable timing of thread execution. Example: If two threads both read balance=100, add 50, and write back, the result might be 150 instead of 200 because both threads read the initial value before either writes.

3.2 Banking Example Structure

```c
// Shared data structure
typedef struct {
    int account_id;
    double balance;
    int transaction_count;
} Account;

// Global accounts array (shared resource)
Account accounts[NUM_ACCOUNTS];

// Thread function
void* teller_thread(void* arg) {
    int teller_id = *(int*)arg;  // Cast void* to int* and dereference

    // Perform multiple transactions
    for (int i = 0; i < TRANSACTIONS_PER_TELLER; i++) {
        // Select random account
        // Perform deposit or withdrawal
        // THIS WILL HAVE RACE CONDITIONS!

        printf("Teller %d: Transaction %d\n", teller_id, i);
    }

    return NULL;
}

// Creating threads (see Appendix \ref{sec:voidpointer} for void* explanation)
pthread_t threads[NUM_THREADS];
int thread_ids[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
    thread_ids[i] = i;
    pthread_create(&threads[i], NULL, teller_thread, &thread_ids[i]);
}
```

3.3 Expected Output

Your Phase 1 should demonstrate race conditions:

```
Initial balance: 1000.00
Thread 1: Depositing 100.00
Thread 2: Depositing 100.00
Thread 3: Withdrawing 50.00
Final balance: 1150.00  // WRONG! Should be 1150.00

// Run again - different result!
Final balance: 1100.00  // Race condition causes inconsistent results
```

4 Phase 2: Resource Protection

Add mutex locks to protect shared resources and eliminate race conditions.

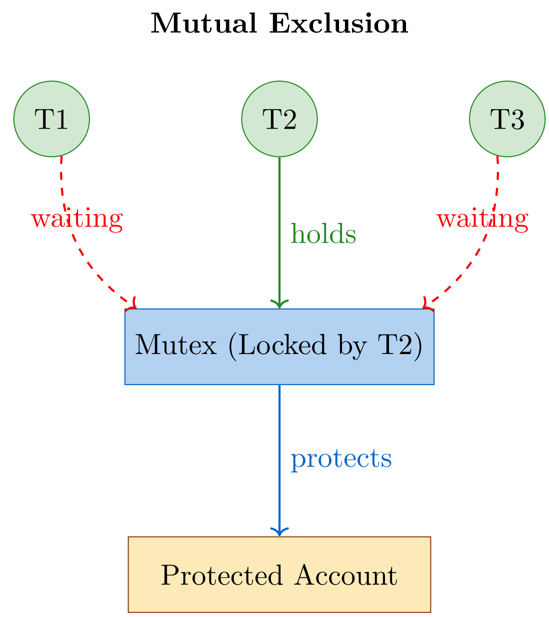


Figure 3: Phase 2: Mutex ensuring only one thread accesses resource at a time

4.1 Requirements

- Implement pthread mutexes for each account
- Ensure thread-safe access to all shared data
- Verify correct final balances
- Measure performance impact of synchronization

4.2 Key Additions

```
// Add mutex to account structure
typedef struct {
    int account_id;
    double balance;
    int transaction_count;
    pthread_mutex_t lock;  // Mutex for this account
} Account;

// Initialize mutexes - MUST be done before creating threads!
for (int i = 0; i < NUM_ACCOUNTS; i++) {
    pthread_mutex_init(&accounts[i].lock, NULL);
    accounts[i].balance = INITIAL_BALANCE;
    accounts[i].transaction_count = 0;
}
```

```c

// Protected transaction
void deposit(int account_id, double amount) {
    pthread_mutex_lock(&accounts[account_id].lock);
    // Critical section - only one thread can execute this at a time
    accounts[account_id].balance += amount;
    accounts[account_id].transaction_count++;
    pthread_mutex_unlock(&accounts[account_id].lock);
}

// Error checking (optional but recommended)
if (pthread_mutex_lock(&accounts[id].lock) != 0) {
    perror("Failed to acquire lock");
    return;
}
```

5 Phase 3: Deadlock Creation

Implement account transfers that can cause deadlock.

5.1 Requirements

- Implement transfer operations requiring two account locks
- Create a scenario where deadlock is highly likely to occur
- Detect and report when threads appear stuck (no progress)
- Use multiple threads performing transfers between same accounts

Creating Reliable Deadlock

Deadlock requires four conditions (Coffman conditions):

1. Mutual Exclusion: Resources cannot be shared
2. Hold and Wait: Thread holds one resource while waiting for another
3. No Preemption: Resources cannot be forcibly taken
4. Circular Wait: Circular chain of threads waiting for resources

Your transfer function should create these conditions!

5.2 Deadlock Scenario

```c
void transfer(int from_id, int to_id, double amount) {
    printf("Thread %ld: Attempting transfer from %d to %d\n",
           pthread_self(), from_id, to_id);

    pthread_mutex_lock(&accounts[from_id].lock);
    printf("Thread %ld: Locked account %d\n", pthread_self(), from_id);

    // Simulate processing delay - gives other thread time to create
       deadlock
    usleep(100);  // Sleep for 100 microseconds

    printf("Thread %ld: Waiting for account %d\n", pthread_self(), to_id);
    pthread_mutex_lock(&accounts[to_id].lock);

    // If we get here, no deadlock occurred this time
    accounts[from_id].balance -= amount;
    accounts[to_id].balance += amount;

    pthread_mutex_unlock(&accounts[to_id].lock);
    pthread_mutex_unlock(&accounts[from_id].lock);
}

// Create threads that will deadlock:
// Thread 1: transfer(A, B)  // Locks A, waits for B
// Thread 2: transfer(B, A)  // Locks B, waits for A
// Result: Both threads wait forever!
```

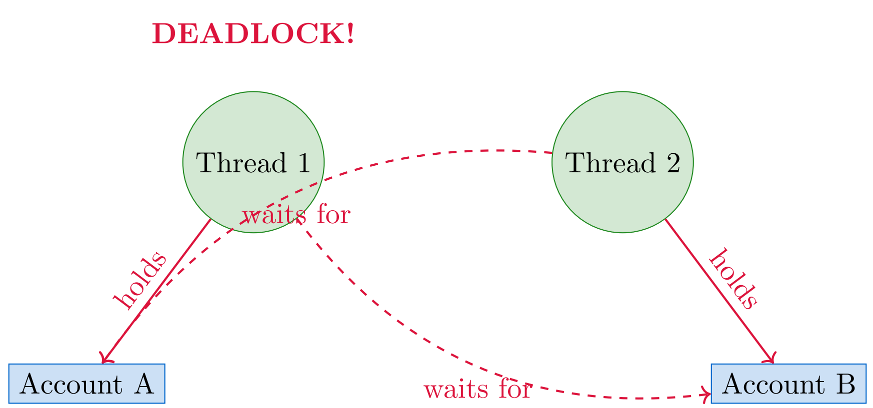


Figure 4: Circular Wait Condition

6 Phase 4: Deadlock Resolution

Implement strategies to prevent or resolve deadlock.

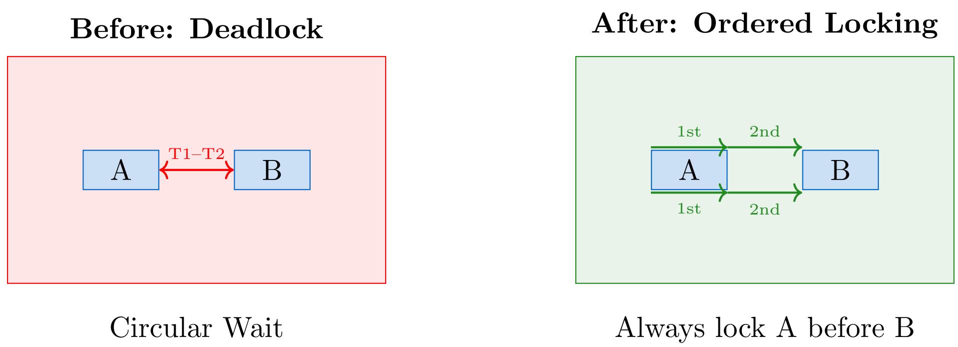


Figure 5: Phase 4: Solving deadlock with ordered resource acquisition

6.1 Requirements

Choose and implement at least ONE strategy:

1. **Lock Ordering**: Always acquire locks in consistent order
2. **Timeout Mechanism**: Use `pthread_mutex_timedlock`
3. **Deadlock Detection**: Implement a detection algorithm
4. **Banker's Algorithm**: Implement deadlock avoidance

6.2 Example: Lock Ordering Solution

```c
void safe_transfer(int from_id, int to_id, double amount) {
    // Always lock lower ID first
    int first = (from_id < to_id) ? from_id : to_id;
    int second = (from_id < to_id) ? to_id : from_id;

    pthread_mutex_lock(&accounts[first].lock);
    pthread_mutex_lock(&accounts[second].lock);

    // Perform transfer
    accounts[from_id].balance -= amount;
    accounts[to_id].balance += amount;

    pthread_mutex_unlock(&accounts[second].lock);
    pthread_mutex_unlock(&accounts[first].lock);
}
```

7 Implementation Guidelines

7.1 Required Headers

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>  // For strerror()
#include <errno.h>   // For error codes
```

7.2 Getting Started Tips

Experimentation Encouraged!

This project is about breaking things and fixing them:

- Try different numbers of threads (2, 5, 10, 100!)
- Experiment with timing delays
- Test edge cases (negative amounts, non-existent accounts)
- Add random sleeps to change thread scheduling
- Run your program many times - race conditions may not appear every time!

7.2.1 Random Numbers per Thread

```c
void* thread_function(void* arg) {
    // Seed random number generator per thread
    unsigned int seed = time(NULL) + pthread_self();

    // Use rand_r() for thread-safe random numbers
    int random_account = rand_r(&seed) % NUM_ACCOUNTS;
}
```

7.2.2 Proper Thread Cleanup

```c
// In main(), after creating all threads:

// Wait for all threads to complete
for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
}

// Clean up mutexes AFTER all threads are done
for (int i = 0; i < NUM_ACCOUNTS; i++) {
    pthread_mutex_destroy(&accounts[i].lock);
```

```
}

// Main thread should be the last to exit
return 0;
```

7.3 Compilation

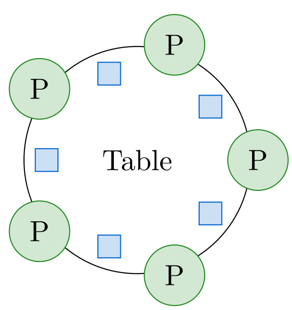
```
gcc -Wall -pthread phase1.c -o phase1
gcc -Wall -pthread phase2.c -o phase2
gcc -Wall -pthread phase3.c -o phase3
gcc -Wall -pthread phase4.c -o phase4
```

8 Alternative Scenarios

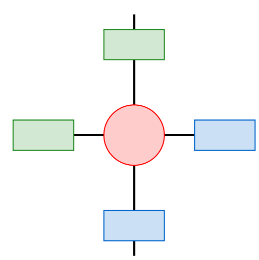
If you prefer not to use the banking example:

- **Dining Philosophers**: Classic deadlock problem
- **Traffic Intersection**: Cars as threads, intersections as resources
- **Resource Allocation**: Threads competing for printers/scanners
- **Database Records**: Threads updating multiple records

The key is demonstrating the four phases clearly.



Alternative: Dining Philosophers



Alternative: Traffic Control

Figure 6: Alternative project scenarios

9 Testing and Validation

9.1 Phase 1 Testing

- Run multiple times and observe different incorrect results

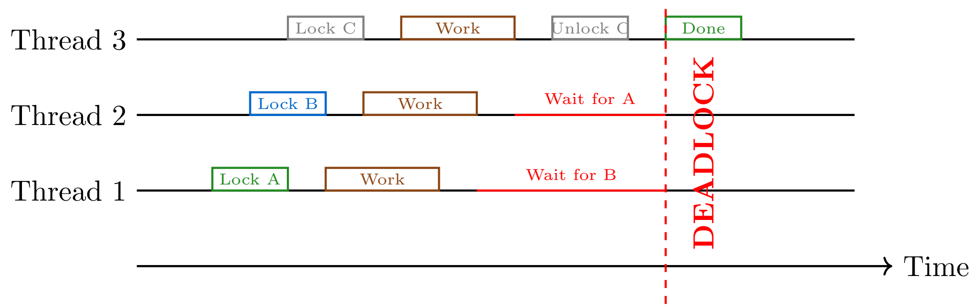


Figure 7: Timeline showing deadlock between Thread 1 and Thread 2

- Log all operations to show interleaving
- Calculate expected vs actual results

9.2 Phase 2 Testing

- Verify final balances are always correct
- Measure execution time with/without locks
- Stress test with many threads

9.3 Phase 3 Testing

- Confirm deadlock occurs reliably
- Use thread monitoring tools
- Add timeout detection to identify stuck threads

9.4 Phase 4 Testing

- Verify no deadlocks occur over many runs
- Compare performance of different strategies
- Test edge cases

10 Deliverables

Submit the following files directly to D2L:

10.1 1. Source Code

- `phase1.c` - Basic threading with race conditions
- `phase2.c` - Mutex-protected version
- `phase3.c` - Deadlock demonstration
- `phase4.c` - Deadlock resolution

- `README.txt` - Brief description of your approach

10.2 2. Documentation

- **Report.pdf** - LaTeX-generated report including:
  - Your approach to each phase
  - Screenshots of program output
  - Challenges encountered and solutions
  - Performance observations
- **Screenshots** showing each phase running

10.3 3. Demonstration Video (Maximum 5 minutes)

Record a video showing:

- Brief introduction (your name, which project variant you chose)
- Phase 1: Race conditions occurring
- Phase 2: Successful synchronization
- Phase 3: Deadlock happening
- Phase 4: Deadlock resolved
- Quick code walkthrough of key sections

Video Recording Tips

- Keep it under 5 minutes - practice your demo first
- Show your terminal clearly with readable font size
- Explain what's happening as you demonstrate
- Focus on results, not line-by-line code reading
- Include your GitHub/GitLab repository URL in the video

10.4 4. Optional: Repository Link

Include your Git repository URL in your report or D2L submission comments.

11 Common Pitfalls

Avoid These Mistakes

- **Forgetting to join threads**: Always use `pthread_join`
- **Not initializing mutexes**: Call `pthread_mutex_init`
- **Forgetting cleanup**: Destroy mutexes when done
- **Stack variables for thread arguments**: Use heap or ensure scope
- **Printf debugging**: Can mask timing issues - use it carefully

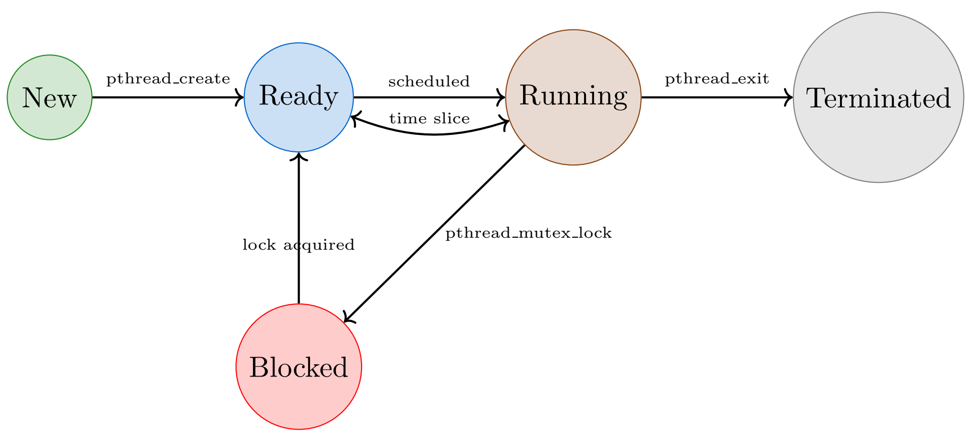


Figure 8: Thread State Transitions

A Quick Reference

A.1 Essential pthread Functions

```
// Thread management
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
// Returns: 0 on success, error number on failure

int pthread_join(pthread_t thread, void **retval);
// Waits for thread to terminate

pthread_t pthread_self(void);
// Returns ID of calling thread

// Mutex operations
```

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

// Advanced mutex operations
int pthread_mutex_trylock(pthread_mutex_t *mutex);  // Non-blocking
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *timeout);
```

A.2 Understanding void* (Generic Pointers)

The `void*` type is a generic pointer that can point to any data type:

```c
// Thread functions must have this signature:
void* thread_function(void* arg);

// Passing an integer to a thread:
int value = 42;
pthread_create(&thread, NULL, thread_function, &value);

// Inside thread function:
void* thread_function(void* arg) {
    int* received_value = (int*)arg;  // Cast to correct type
    printf("Received: %d\n", *received_value);  // Dereference
    return NULL;
}

// Passing a struct:
typedef struct {
    int id;
    char name[50];
} ThreadData;

ThreadData data = {1, "Thread One"};
pthread_create(&thread, NULL, thread_function, &data);

// Inside thread function:
void* thread_function(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    printf("ID: %d, Name: %s\n", data->id, data->name);
    return NULL;
}
```

Warning: Be careful with stack variables! This is WRONG:

```c
for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&threads[i], NULL, thread_func, &i);
    // Bug: 'i' changes before thread reads it!
}
```

A.3 Common Functions Reference

`usleep(microseconds)` Suspends thread execution for specified microseconds (1 second = 1,000,000 microseconds)

`sleep(seconds)` Suspends thread execution for specified seconds

`rand_r(&seed)` Thread-safe random number generator (unlike `rand()`)

`pthread_self()` Returns the thread ID of the calling thread

`perror("message")` Prints error message with description of last error

A.4 Debugging Deadlocks

Signs your program is deadlocked:

- Program stops producing output
- CPU usage drops to near 0%
- Threads show as "waiting" in system monitor

Detection techniques:

```c
// Add timeout detection
time_t start = time(NULL);
while (trying_to_lock) {
    if (time(NULL) - start > 5) {
        printf("Possible deadlock detected!\n");
        break;
    }
}
```

A.5 Useful Debugging Commands

```bash
# Monitor threads in another terminal
ps -eLf | grep your_program

# Use GDB for thread debugging
gdb ./your_program
(gdb) run
(gdb) info threads         # When deadlocked, shows all threads
(gdb) thread 2            # Switch to thread 2
(gdb) where               # Show where thread is stuck

# Detect race conditions and deadlocks
valgrind --tool=helgrind ./your_program
```