# Operating Systems Lab Assignment: Synchronization and Scheduling

Kendyn Fredieu

October 23, 2025

## 1 Introduction

This report documents the implementations and analyses for the synchronization and scheduling lab assignment, covering five provided problems and four additional exercises using mutexes and condition variables.

## 2 Exercise 1: Hello World

```c
// hello_sync.c
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t lock;
pthread_cond_t  cv;
int hello = 0;

void* print_hello(void* arg) {
    pthread_mutex_lock(&lock);
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);         // signal AFTER updating the state
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t thread;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);

    // Lock BEFORE creating the child to prevent a lost signal.
    pthread_mutex_lock(&lock);

    pthread_create(&thread, NULL, print_hello, NULL);

    // Wait in a loop to handle spurious wakeups.
    while (hello < 1) {
        pthread_cond_wait(&cv, &lock);  // atomically unlocks & waits, then re-
            locks
    }

    printf("Second line (hello=%d)\n", hello);
    pthread_mutex_unlock(&lock);

    pthread_join(thread, NULL);
    pthread_cond_destroy(&cv);
```

```
    pthread_mutex_destroy(&lock);
    return 0;
}
```

**Explanation**: Explain why the original code fails and how mutexes and condition variables fix it.
**Analysis**: Discuss synchronization behavior.
**Screenshot**: Include a screenshot of compiling and running hello_world.c.

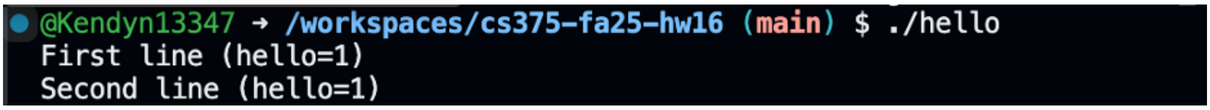


Figure 1: Compilation and execution of hello_world.c

## 3 Exercise 2: SpaceX Problems

```c
// spacex_countdown.c
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv   = PTHREAD_COND_INITIALIZER;
int n = 3;

void* counter(void* arg) {
    pthread_mutex_lock(&lock);
    while (n > 0) {
        printf("%d\n", n);
        n--;
        pthread_cond_signal(&cv);   // wake announcer in case it's waiting
        // keep the lock so prints are serialized; loop will re-check n
    }
    pthread_mutex_unlock(&lock);
    return NULL;
}

void* announcer(void* arg) {
    pthread_mutex_lock(&lock);
    while (n != 0) {                // wait until countdown reaches zero
        pthread_cond_wait(&cv, &lock);
    }
    printf("FALCON HEAVY TOUCH DOWN!\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t t_counter, t_ann;

    // Start announcer first so it likely waits before first signal
    pthread_create(&t_ann,     NULL, announcer, NULL);
    pthread_create(&t_counter, NULL, counter,   NULL);

    pthread_join(t_counter, NULL);
    pthread_join(t_ann, NULL);
    return 0;
}
```

**Explanation**: Describe the original issue and the synchronization fix.
**Analysis**: Analyze countdown and announcement behavior.
**Screenshot**: Include a screenshot of compiling and running spacex.c.

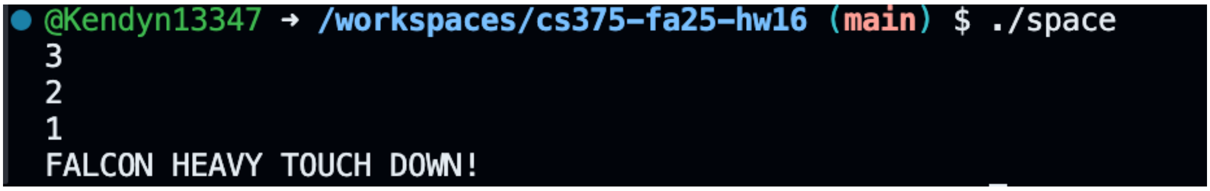


Figure 2: Compilation and execution of spacex.c

## 4 Exercise 3: I Love You, Unconditionally!

```c
// love.c
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t lock;
pthread_cond_t  cv;
int subaru = 0;

void* helper(void* arg) {
    pthread_mutex_lock(&lock);
    subaru += 1;                 // update the predicate under the lock
    pthread_cond_signal(&cv);    // signal AFTER changing the state
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t thread;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);

    // Lock before creating the thread to avoid any chance of a lost wakeup
    pthread_mutex_lock(&lock);
    pthread_create(&thread, NULL, helper, NULL);

    // Mesa semantics: wait in a WHILE loop guarding the predicate
    while (subaru != 1) {
        pthread_cond_wait(&cv, &lock);  // atomically releases & re-acquires the lock
    }

    printf("I love Emilia!\n");         // runs only after helper has incremented subaru
    pthread_mutex_unlock(&lock);

    pthread_join(thread, NULL);
    pthread_cond_destroy(&cv);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

**Explanation**: Explain how condition variables ensure correct output.
**Analysis**: Discuss race condition prevention.
**Screenshot**: Include a screenshot of compiling and running love.c.

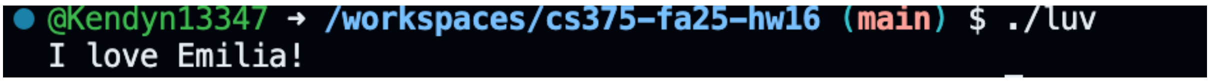


Figure 3: Compilation and execution of love.c

## 5 Exercise 4: Locking Up the Floopies

```c
// floopy.c
#include <pthread.h>
#include <stdio.h>

typedef struct account {
    pthread_mutex_t lock;
    int    balance;
    long   uuid;
} account_t;

typedef struct {
    account_t* donor;
    account_t* recipient;
    int amount;
} transfer_args_t;

/* Lock-ordering transfer: always acquire locks in ascending uuid order. */
void transfer(account_t* donor, account_t* recipient, int amount) {
    account_t* first  = (donor->uuid < recipient->uuid) ? donor     : recipient
        ;
    account_t* second = (donor->uuid < recipient->uuid) ? recipient : donor;

    pthread_mutex_lock(&first->lock);
    pthread_mutex_lock(&second->lock);

    if (donor->balance < amount) {
        printf("Insufficient␣funds.\n");
    } else {
        donor->balance    -= amount;
        recipient->balance += amount;
        printf("Transferred␣%d␣from␣account␣%ld␣to␣%ld\n",
               amount, donor->uuid, recipient->uuid);
    }

    pthread_mutex_unlock(&second->lock);
    pthread_mutex_unlock(&first->lock);
}

void* transfer_thread(void* arg) {
    transfer_args_t* a = (transfer_args_t*)arg;
    transfer(a->donor, a->recipient, a->amount);
    return NULL;
}

int main(void) {
    account_t acc1 = { PTHREAD_MUTEX_INITIALIZER, 1000, 1 };
    account_t acc2 = { PTHREAD_MUTEX_INITIALIZER,  500, 2 };

    transfer_args_t t1 = { &acc1, &acc2, 200 };  // A -> B
    transfer_args_t t2 = { &acc2, &acc1, 100 };  // B -> A

    pthread_t th1, th2;
    pthread_create(&th1, NULL, transfer_thread, &t1);
    pthread_create(&th2, NULL, transfer_thread, &t2);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    return 0;
}
```

**Explanation**: Describe the deadlock issue and lock ordering solution.

**Analysis**: Provide a deadlock scenario and analyze the fix.
**Screenshot**: Include a screenshot of compiling and running floopy.c.

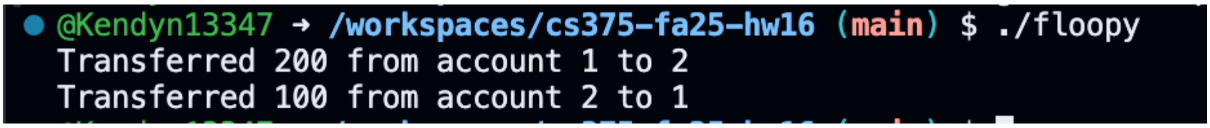


Figure 4: Compilation and execution of floopy.c

# 6 Exercise 5: Baking with Condition Variables

```
// baking.c
#include <pthread.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>

int  numBatterInBowl = 0;
int  numEggInBowl    = 0;
bool readyToEat      = false;

pthread_mutex_t lock;
pthread_cond_t  needIngredients;  // wake ingredient threads to add more
pthread_cond_t  readyToBake;      // wake heater when bowl has 1 batter + 2
    eggs
pthread_cond_t  startEating;      // wake eater when cake is ready

static void addBatter(void) { numBatterInBowl += 1; }
static void addEgg(void)    { numEggInBowl    += 1; }
static void heatBowl(void)  { readyToEat = true; numBatterInBowl = 0;
    numEggInBowl = 0; }
static void eatCake(void)   { readyToEat = false; }

void* batterAdder(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        // only one batter per cake, and don't add during/after baking
        while (numBatterInBowl != 0 || readyToEat) {
            pthread_cond_wait(&needIngredients, &lock);
        }
        addBatter();
        printf("[Batter]␣+1␣(batter=%d,␣eggs=%d)\n", numBatterInBowl,
            numEggInBowl);
        pthread_cond_signal(&readyToBake);  // heater may proceed if eggs ready
        pthread_mutex_unlock(&lock);
        pthread_mutex_lock(&lock);
    }
}

void* eggBreaker(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        // need exactly two eggs; don't add during/after baking
        while (numEggInBowl >= 2 || readyToEat) {
            pthread_cond_wait(&needIngredients, &lock);
        }
        addEgg();
        printf("[Egg]␣+1␣(batter=%d,␣eggs=%d)\n", numBatterInBowl, numEggInBowl
            );
```

```
        pthread_cond_signal(&readyToBake);  // heater may proceed if batter
            present
        pthread_mutex_unlock(&lock);
        pthread_mutex_lock(&lock);
    }
}

void* bowlHeater(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        // bake only when ingredients are ready and not already eating
        while (numBatterInBowl < 1 || numEggInBowl < 2 || readyToEat) {
            pthread_cond_wait(&readyToBake, &lock);
        }
        printf("[Heater]␣Baking...\n");
        heatBowl();
        printf("[Heater]␣Cake␣ready!␣(batter=%d,␣eggs=%d)\n", numBatterInBowl,
            numEggInBowl);
        pthread_cond_signal(&startEating);  // let eater proceed
        pthread_mutex_unlock(&lock);
        pthread_mutex_lock(&lock);
    }
}

void* cakeEater(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        while (!readyToEat) {
            pthread_cond_wait(&startEating, &lock);
        }
        printf("[Eater]␣Eating␣cake!\n");
        eatCake();
        printf("[Eater]␣Done.␣Requesting␣more␣ingredients.\n");
        // wake ALL ingredient threads to start the next cycle
        pthread_cond_broadcast(&needIngredients);
        pthread_mutex_unlock(&lock);
        pthread_mutex_lock(&lock);
    }
}

int main(void) {
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&needIngredients, NULL);
    pthread_cond_init(&readyToBake, NULL);
    pthread_cond_init(&startEating, NULL);

    pthread_t batter, egg1, egg2, heater, eater;
    pthread_create(&batter, NULL, batterAdder, NULL);
    pthread_create(&egg1,  NULL, eggBreaker, NULL);
    pthread_create(&egg2,  NULL, eggBreaker, NULL);
    pthread_create(&heater,NULL, bowlHeater,  NULL);
    pthread_create(&eater, NULL, cakeEater,   NULL);

    // keep the process alive
    while (1) sleep(1);
    return 0;
}
```

**Explanation**: Explain how condition variables coordinate baking steps.
**Analysis**: Discuss thread synchronization.
**Screenshot**: Include a screenshot of compiling and running baking.c.

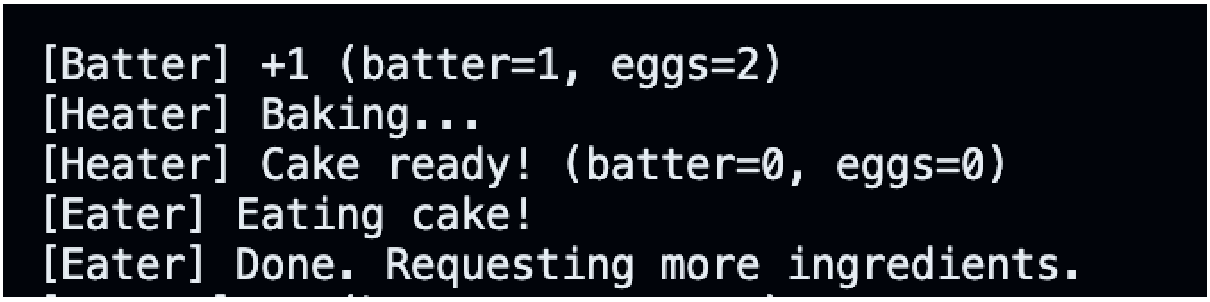


Figure 5: Compilation and execution of baking.c

# 7 Exercise 6: Priority Donation in Transfer

```c
// priority_transfer.c
// Build: gcc -pthread priority_transfer.c -o priority_transfer
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

typedef struct {
    pthread_mutex_t m;
    pthread_t owner;
    int owner_base_prio;   // owner's original/base priority
    int owner_eff_prio;    // owner's effective (possibly boosted) priority
} prio_mutex_t;

static void prio_mutex_init(prio_mutex_t* pm) {
    pthread_mutex_init(&pm->m, NULL);
    pm->owner = 0;
    pm->owner_base_prio = pm->owner_eff_prio = 0;
}

static void prio_mutex_unlock(prio_mutex_t* pm) {
    // On unlock, drop effective priority back to base.
    pm->owner_eff_prio = pm->owner_base_prio;
    pm->owner = 0;
    pthread_mutex_unlock(&pm->m);
}

// Trylock; if busy, "donate" caller_prio to current owner, then block.
static void prio_mutex_lock(prio_mutex_t* pm, int caller_prio) {
    if (pthread_mutex_trylock(&pm->m) == 0) {
        pm->owner = pthread_self();
        pm->owner_base_prio = caller_prio;
        pm->owner_eff_prio  = caller_prio;
        return;
    }
    // Already locked: simulate priority donation
    // (We can't query the real owner's thread priority, so we store it.)
    if (caller_prio > pm->owner_eff_prio) {
        pm->owner_eff_prio = caller_prio;
        fprintf(stderr,
                "[donate] T%lu donates prio %d to owner T%lu (eff=%d)\n",
                (unsigned long)pthread_self(), caller_prio,
                (unsigned long)pm->owner, pm->owner_eff_prio);
    }
    // Block until available; when we acquire, we become the owner.
    pthread_mutex_lock(&pm->m);
```

```c
    pm->owner = pthread_self();
    pm->owner_base_prio = caller_prio;
    pm->owner_eff_prio  = caller_prio;
}

// ---- Accounts & transfer ----
typedef struct account {
    prio_mutex_t lock;
    int    balance;
    long   uuid;
} account_t;

typedef struct {
    account_t* donor;
    account_t* recipient;
    int amount;
    int thread_prio;    // simulated priority (higher = more important)
    const char* name;
} transfer_args_t;

// Busy work to exaggerate inversion windows
static void burn_cpu_ms(int ms) {
    usleep(ms * 1000);
}

// Always lock in ascending-uuid order + simulate donation while waiting.
static void transfer(account_t* donor, account_t* recipient, int amount, int
    thr_prio, const char* who) {
    account_t* first  = (donor->uuid < recipient->uuid) ? donor     : recipient
        ;
    account_t* second = (donor->uuid < recipient->uuid) ? recipient : donor;

    // Acquire first lock
    prio_mutex_lock(&first->lock, thr_prio);
    // Simulate some work while holding first lock to create contention
    burn_cpu_ms(30);

    // Acquire second lock (donation may occur inside prio_mutex_lock)
    prio_mutex_lock(&second->lock, thr_prio);

    if (donor->balance < amount) {
        printf("[%s] Insufficient funds.\n", who);
    } else {
        donor->balance     -= amount;
        recipient->balance += amount;
        printf("[%s] Transferred %d from %ld to %ld (balances: %d, %d)\n",
               who, amount, donor->uuid, recipient->uuid, donor->balance,
                   recipient->balance);
    }

    prio_mutex_unlock(&second->lock);
    prio_mutex_unlock(&first->lock);
}

static void* transfer_thread(void* arg) {
    transfer_args_t* a = (transfer_args_t*)arg;

    // Each thread does two transfers to amplify interaction.
    transfer(a->donor, a->recipient, a->amount, a->thread_prio, a->name);
    burn_cpu_ms(20);
    transfer(a->recipient, a->donor, a->amount / 2, a->thread_prio, a->name);
    return NULL;
}
```

```c
int main(void) {
    account_t A, B;
    prio_mutex_init(&A.lock);
    prio_mutex_init(&B.lock);
    A.balance = 1000; A.uuid = 1;
    B.balance =  500; B.uuid = 2;

    transfer_args_t hi = { &A, &B, 200, /*prio*/ 90, "HIGH" };
    transfer_args_t lo = { &B, &A, 100, /*prio*/ 10, "LOW"  };

    pthread_t th_hi, th_lo;

    // Start LOW first so it grabs one lock and then blocks on the other.
    pthread_create(&th_lo, NULL, transfer_thread, &lo);
    burn_cpu_ms(5);
    pthread_create(&th_hi, NULL, transfer_thread, &hi);

    pthread_join(th_hi, NULL);
    pthread_join(th_lo, NULL);

    printf("Final balances: A=%d B=%d\n", A.balance, B.balance);
    return 0;
}
```

**Explanation**: Describe how priority donation prevents priority inversion.
**Analysis**: Analyze priority handling.
**Screenshot**: Include a screenshot of compiling and running priority_transfer.c.

```
● @Kendyn13347 → /workspaces/cs375-fa25-hw16 (main) $ ./priority
  [donate] T129398644901568 donates prio 90 to owner T129398653294272 (eff=90)
  [LOW] Transferred 100 from 2 to 1 (balances: 400, 1100)
  [HIGH] Transferred 200 from 1 to 2 (balances: 900, 600)
  [donate] T129398644901568 donates prio 90 to owner T129398653294272 (eff=90)
  [LOW] Transferred 50 from 1 to 2 (balances: 850, 650)
  [HIGH] Transferred 100 from 2 to 1 (balances: 550, 950)
  Final balances: A=950 B=550
```

Figure 6: Compilation and execution of priority_transfer.c

## 8 Exercise 7: Barrier Synchronization

```c
// barrier.c
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 4

pthread_mutex_t lock;
pthread_cond_t  cv;
int count = 0;
int generation = 0;   // increments each time the barrier opens

void barrier(void) {
    pthread_mutex_lock(&lock);

    int my_gen = generation;      // snapshot the current "sense"
    if (++count == NUM_THREADS) {
        // Last thread arrives: open the barrier for everyone
        count = 0;                // reset for next use
        generation++;             // flip sense
```

```c
        pthread_cond_broadcast(&cv);
    } else {
        // Wait until the generation changes
        while (my_gen == generation) {
            pthread_cond_wait(&cv, &lock);
        }
    }

    pthread_mutex_unlock(&lock);
}

void* worker(void* arg) {
    int id = *(int*)arg;

    printf("Thread %d: Before barrier 1\n", id);
    barrier();
    printf("Thread %d: After  barrier 1\n", id);

    // Demonstrate reusability:
    printf("Thread %d: Before barrier 2\n", id);
    barrier();
    printf("Thread %d: After  barrier 2\n", id);

    return NULL;
}

int main(void) {
    pthread_t threads[NUM_THREADS];
    int ids[NUM_THREADS];

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);

    for (int i = 0; i < NUM_THREADS; ++i) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, worker, &ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }

    pthread_cond_destroy(&cv);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

**Explanation**: Explain how the barrier synchronizes threads.
**Analysis**: Discuss barrier behavior.
**Screenshot**: Include a screenshot of compiling and running barrier.c.

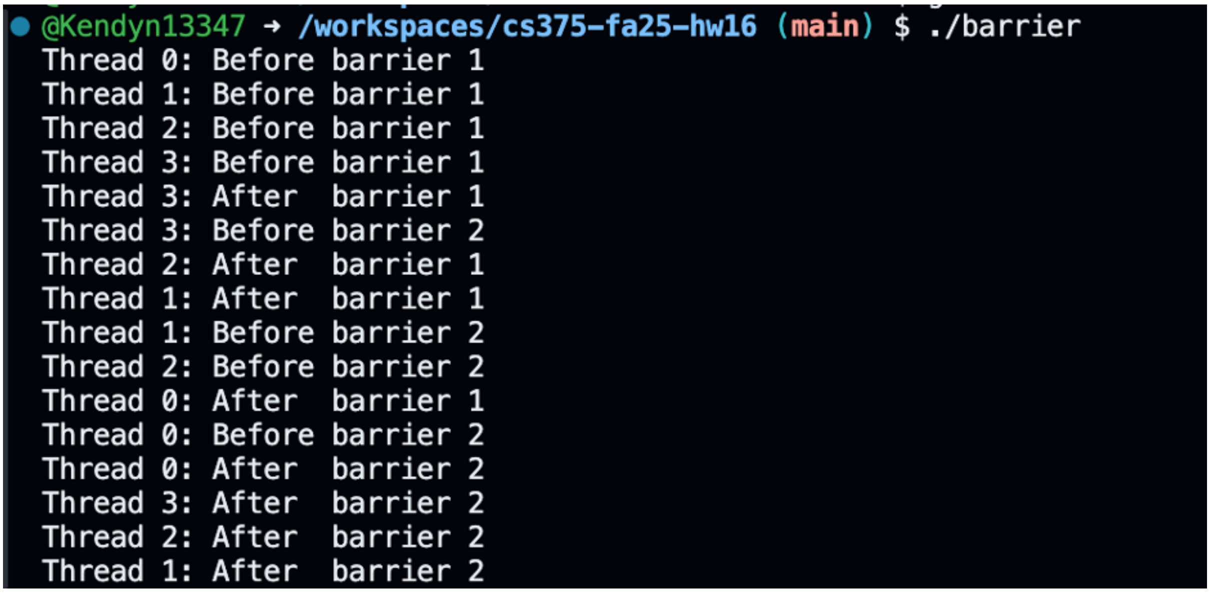


Figure 7: Compilation and execution of barrier.c

## 9 Exercise 8: Readers-Writers with Priority

```c
// readers_writers.c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_mutex_t lock;
pthread_cond_t  reader_cv, writer_cv;

int reader_count   = 0;  // number of active readers
int writer_waiting = 0;  // number of writers waiting
int writer_active  = 0;  // 1 if a writer holds the resource
int shared_data    = 0;

void* reader(void* arg) {
    // ENTRY
    pthread_mutex_lock(&lock);
    while (writer_waiting > 0 || writer_active) {   // writers have priority
        pthread_cond_wait(&reader_cv, &lock);
    }
    reader_count++;
    pthread_mutex_unlock(&lock);

    // CRITICAL SECTION (read)
    printf("Reader reads: %d\n", shared_data);
    usleep(50 * 1000);

    // EXIT
    pthread_mutex_lock(&lock);
    reader_count--;
    if (reader_count == 0)
        pthread_cond_signal(&writer_cv);            // let a waiting writer go
    pthread_mutex_unlock(&lock);
    return NULL;
}

void* writer(void* arg) {
    // ENTRY
```

```c
    pthread_mutex_lock(&lock);
    writer_waiting++;
    while (reader_count > 0 || writer_active) {
        pthread_cond_wait(&writer_cv, &lock);
    }
    writer_waiting--;
    writer_active = 1;
    pthread_mutex_unlock(&lock);

    // CRITICAL SECTION (write)
    shared_data++;
    printf("Writer writes: %d\n", shared_data);
    usleep(60 * 1000);

    // EXIT
    pthread_mutex_lock(&lock);
    writer_active = 0;
    if (writer_waiting > 0) {
        pthread_cond_signal(&writer_cv);          // next writer first (priority)
    } else {
        pthread_cond_broadcast(&reader_cv);       // otherwise free all readers
    }
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t readers[3], writers[2];

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&reader_cv, NULL);
    pthread_cond_init(&writer_cv, NULL);

    for (int i = 0; i < 3; ++i) pthread_create(&readers[i], NULL, reader, NULL);
    for (int i = 0; i < 2; ++i) pthread_create(&writers[i], NULL, writer, NULL);
    for (int i = 0; i < 3; ++i) pthread_join(readers[i], NULL);
    for (int i = 0; i < 2; ++i) pthread_join(writers[i], NULL);

    pthread_cond_destroy(&reader_cv);
    pthread_cond_destroy(&writer_cv);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

**Explanation**: Describe writer priority enforcement.
**Analysis**: Analyze reader-writer interactions.
**Screenshot**: Include a screenshot of compiling and running readers_writers.c.

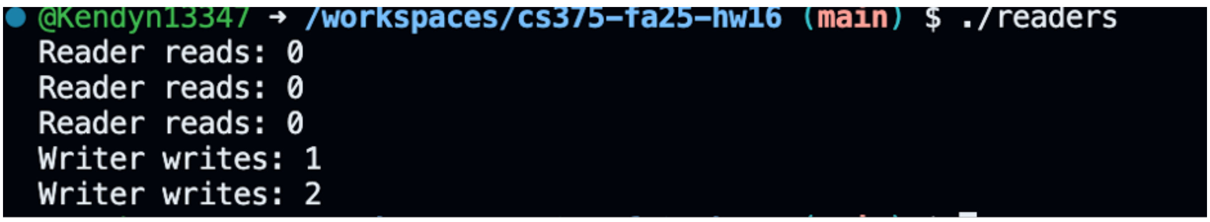


Figure 8: Compilation and execution of readers_writers.c

# 10 Exercise 9: Thread Pool

```c
// thread_pool.c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 4

typedef struct {
    void (*task)(int);
    int arg;
} Task;

typedef struct {
    Task* queue;
    int head, tail, count, size;
    int closed;                         // set when we stop accepting work
    pthread_mutex_t lock;
    pthread_cond_t  not_empty, not_full;
} ThreadSafeQueue;

/* ------------ Queue --------------- */
void queue_init(ThreadSafeQueue* q, int size) {
    q->queue = (Task*)malloc(sizeof(Task) * size);
    q->head = q->tail = q->count = 0;
    q->size = size;
    q->closed = 0;
    pthread_mutex_init(&q->lock, NULL);
    pthread_cond_init(&q->not_empty, NULL);
    pthread_cond_init(&q->not_full, NULL);
}

void queue_destroy(ThreadSafeQueue* q) {
    pthread_mutex_destroy(&q->lock);
    pthread_cond_destroy(&q->not_empty);
    pthread_cond_destroy(&q->not_full);
    free(q->queue);
}

void queue_close(ThreadSafeQueue* q) {
    pthread_mutex_lock(&q->lock);
    q->closed = 1;
    pthread_cond_broadcast(&q->not_empty);  // wake waiting workers
    pthread_cond_broadcast(&q->not_full);   // wake producers, if any
    pthread_mutex_unlock(&q->lock);
}

int queue_push(ThreadSafeQueue* q, Task task) {
    pthread_mutex_lock(&q->lock);
    while (!q->closed && q->count == q->size) {
        pthread_cond_wait(&q->not_full, &q->lock);
    }
    if (q->closed) {                        // no longer accepting tasks
        pthread_mutex_unlock(&q->lock);
        return 0;
    }
    q->queue[q->tail] = task;
    q->tail = (q->tail + 1) % q->size;
    q->count++;
    pthread_cond_signal(&q->not_empty);
    pthread_mutex_unlock(&q->lock);
```

```c
    return 1;
}

/* returns 1 if a task was popped, 0 if queue is closed and empty */
int queue_pop(ThreadSafeQueue* q, Task* task) {
    pthread_mutex_lock(&q->lock);
    while (q->count == 0 && !q->closed) {
        pthread_cond_wait(&q->not_empty, &q->lock);
    }
    if (q->count == 0 && q->closed) {
        pthread_mutex_unlock(&q->lock);
        return 0;                           // shutdown signal
    }
    *task = q->queue[q->head];
    q->head = (q->head + 1) % q->size;
    q->count--;
    pthread_cond_signal(&q->not_full);
    pthread_mutex_unlock(&q->lock);
    return 1;
}

/* ------------ Thread pool workers --------------- */
void sample_task(int arg) {
    printf("Task executed with arg: %d\n", arg);
    usleep(50000); // simulate some work (~50ms)
}

void* worker(void* arg) {
    ThreadSafeQueue* q = (ThreadSafeQueue*)arg;
    Task t;
    while (queue_pop(q, &t)) {
        t.task(t.arg);
    }
    return NULL;
}

int main(void) {
    ThreadSafeQueue q;
    queue_init(&q, 16);

    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_create(&threads[i], NULL, worker, &q);
    }

    // Enqueue some work
    for (int i = 0; i < 20; ++i) {
        queue_push(&q, (Task){ sample_task, i });
    }

    // No more tasks: close the queue and join workers
    queue_close(&q);
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], NULL);
    }
    queue_destroy(&q);
    return 0;
}
```

**Explanation**: Explain how the thread pool uses a thread-safe queue.
**Analysis**: Discuss efficiency benefits.
**Screenshot**: Include a screenshot of compiling and running thread_pool.c.

```
● @Kendyn13347 → /workspaces/cs375-fa25-hw16 (main) $ ./thread
Task executed with arg: 1
Task executed with arg: 0
Task executed with arg: 2
Task executed with arg: 3
Task executed with arg: 4
Task executed with arg: 5
Task executed with arg: 6
Task executed with arg: 7
Task executed with arg: 8
Task executed with arg: 9
Task executed with arg: 10
Task executed with arg: 11
Task executed with arg: 12
Task executed with arg: 13
Task executed with arg: 14
Task executed with arg: 15
Task executed with arg: 16
Task executed with arg: 17
Task executed with arg: 18
Task executed with arg: 19
```

Figure 9: Compilation and execution of thread_pool.c