Operating Systems Lab Assignment: Synchronization and Scheduling

Cameron Cleveland

October 24, 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.

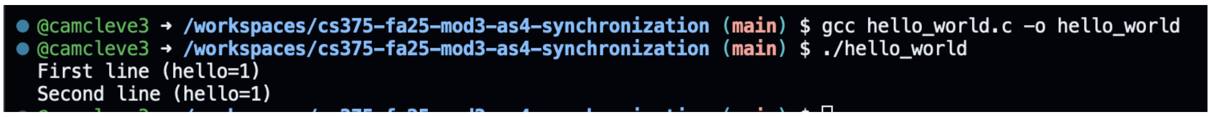
2 Exercise 1: Hello World

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main(void) {
    pthread_t thread;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        perror("pthread_mutex_init");
        return 1;
    }
    if (pthread_cond_init(&cv, NULL) != 0) {
        perror("pthread_cond_init");
        pthread_mutex_destroy(&lock);
        return 1;
    }

    if (pthread_create(&thread, NULL, print_hello, NULL) != 0) {
        perror("pthread_create");
        pthread_cond_destroy(&cv);
        pthread_mutex_destroy(&lock);
        return 1;
    }

    pthread_mutex_lock(&lock);
    while (hello < 1) {
        pthread_cond_wait(&cv, &lock);
    }
    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: The original code had a race condition because the main thread could check or print before the child updated the shared flag. Adding a mutex and condition variable fixes this by letting the child safely update the flag while holding the lock, then signal the main thread. The main thread waits until the flag actually changes, so the output happens in the right order.

Analysis: This one really helped me see how condition variables work. They're basically a clean way to "sleep" until something truly happens. The mutex keeps things atomic, so no one sees half-updated data. I also learned to always wait inside a loop with the lock, so I don't miss signals or react to false ones.

Screenshot: Include a screenshot of compiling and running hello_world.c.



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc hello_world.c -o hello_world
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./hello_world
  First line (hello=1)
  Second line (hello=1)
```

Figure 1: Compilation and execution of hello_world.c

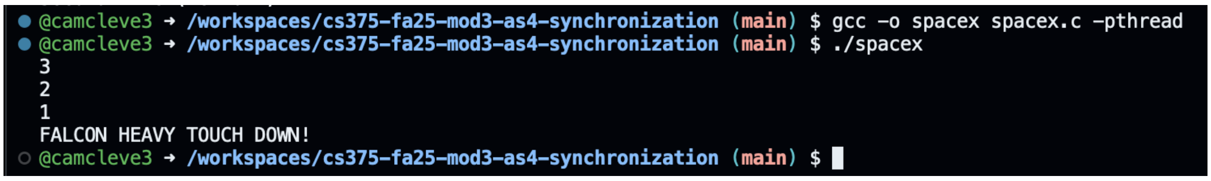
3 Exercise 2: SpaceX Problems

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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);
        fflush(stdout);
        n--;
        pthread_cond_signal(&cv);      // wake announcer each decrement
        pthread_mutex_unlock(&lock);
        usleep(10000);                 // small delay to allow scheduling
        pthread_mutex_lock(&lock);
    }
    // ensure announcer is woken if waiting for n == 0
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
    return NULL;
}

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

int main(void) {
    pthread_t t1, t2;

    if (pthread_create(&t1, NULL, counter, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }
    if (pthread_create(&t2, NULL, announcer, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

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

Explanation: The announcer thread ran too early because it wasn't synced with the countdown. Using a mutex to protect the shared counter and a condition variable to make the announcer wait until it hits zero keeps everything in order. Once the countdown finishes, it signals the announcer to go.

Analysis: The main lesson here: only signal when the actual state changes, and always check that state under the mutex. Doing that guarantees the "launch" announcement happens right after the final countdown, never before.

Screenshot:



```
● @camcleve3 ➜ /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o spacex spacex.c -pthread
● @camcleve3 ➜ /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./spacex
  3
  2
  1
  FALCON HEAVY TOUCH DOWN!
○ @camcleve3 ➜ /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ 
```

Figure 2: Compilation and execution of spacex.c

4 Exercise 3: I Love You, Unconditionally!

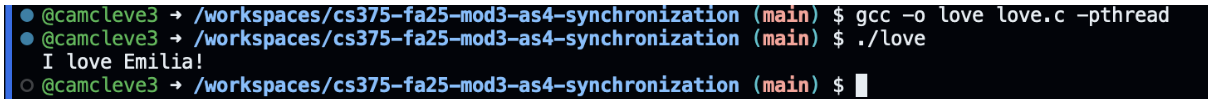
```
// ...existing code...
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

pthread_mutex_t lock;
pthread_cond_t cv;
int subaru = 0;

void* helper(void* arg) {
    pthread_mutex_lock(&lock);
    subaru = 1;
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
    return NULL;
```

```c
}

int main(void) {
    pthread_t thread;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        perror("pthread_mutex_init");
        return 1;
    }
    if (pthread_cond_init(&cv, NULL) != 0) {
        perror("pthread_cond_init");
        pthread_mutex_destroy(&lock);
        return 1;
    }

    if (pthread_create(&thread, NULL, helper, NULL) != 0) {
        perror("pthread_create");
        pthread_cond_destroy(&cv);
        pthread_mutex_destroy(&lock);
        return 1;
    }

    pthread_mutex_lock(&lock);
    while (subaru != 1) {
        pthread_cond_wait(&cv, &lock);
    }
    if (subaru == 1) {
        printf("I␣love␣Emilia!\n");
    } else {
        printf("I␣love␣Rem!\n");
    }
    pthread_mutex_unlock(&lock);

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

Explanation: The main thread needed to confirm that the helper thread really incremented the variable before printing the message. By locking, updating, and signaling from the helper, and having the main wait for that condition, we enforce a clean sequence.

Analysis: This showed me how Mesa-style condition variables actually behave. A signal doesn't mean "it's true now," it means "go check again." You always recheck the condition under the mutex to avoid race conditions and misleading wakeups.

Screenshot:



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o love love.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./love
I love Emilia!
○ @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $
```

Figure 3: Compilation and execution of love.c

5 Exercise 4: Locking Up the Floopies

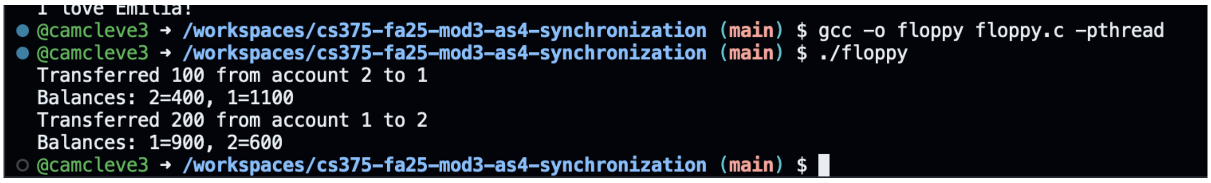
```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

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

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

void transfer(account_t *donor, account_t *recipient, int amount) {
    account_t *first = donor;
    account_t *second = recipient;

    /* consistent ordering: by uuid, break ties by pointer */
    if (donor->uuid > recipient->uuid ||
        (donor->uuid == recipient->uuid && donor > recipient)) {
        first = recipient;
        second = donor;
    }

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

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

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

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

int main(void) {
    account_t acc1, acc2;
    pthread_t t1, t2;
    transfer_args a1, a2;

    acc1.balance = 1000;
    acc1.uuid = 1;
    pthread_mutex_init(&acc1.lock, NULL);

    acc2.balance = 500;
    acc2.uuid = 2;
    pthread_mutex_init(&acc2.lock, NULL);

    a1.donor = &acc1; a1.recipient = &acc2; a1.amount = 200;
    a2.donor = &acc2; a2.recipient = &acc1; a2.amount = 100;

    pthread_create(&t1, NULL, thread_transfer, &a1);
    pthread_create(&t2, NULL, thread_transfer, &a2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&acc1.lock);
    pthread_mutex_destroy(&acc2.lock);
    return 0;
}
```

Explanation: The deadlock came from inconsistent lock ordering: two threads could each grab one account lock and freeze waiting for the other. Fixing that means deciding on a global order, like by account ID, and always locking in that sequence.

Analysis: It's a simple rule but it makes all the difference. Consistent lock order removes cycles in the resource graph, which removes deadlocks. After that, all transfers can run at once without freezing the program.

Screenshot:



```
  I love Emilia!
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o floppy floppy.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./floppy
  Transferred 100 from account 2 to 1
  Balances: 2=400, 1=1100
  Transferred 200 from account 1 to 2
  Balances: 1=900, 2=600
○ @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ 
```

Figure 4: Compilation and execution of floopy.c

6 Exercise 5: Baking with Condition Variables

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>

#define ACTION_DELAY_US 200000  /* microseconds delay after each action */

int numBatterInBowl = 0;
int numEggInBowl = 0;
bool readyToEat = false;
pthread_mutex_t lock;
pthread_cond_t needIngredients, readyToBake, startEating;

void addBatter() {
    numBatterInBowl += 1;
    printf("Added␣batter␣(batter=%d)\n", numBatterInBowl);
    usleep(ACTION_DELAY_US);
}
void addEgg() {
    numEggInBowl += 1;
    printf("Added␣egg␣(eggs=%d)\n", numEggInBowl);
    usleep(ACTION_DELAY_US);
}
void heatBowl() {
    printf("Heating␣bowl\n");
    /* heating sets cake ready */
    readyToEat = true;
    usleep(ACTION_DELAY_US);
}
void eatCake() {
    printf("Eating␣cake\n");
    usleep(ACTION_DELAY_US);
}

/* batter adder: adds 1 batter when bowl needs ingredients */
void* batterAdder(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        while (numBatterInBowl >= 1 || readyToEat) {
            pthread_cond_wait(&needIngredients, &lock);
        }
        addBatter();
        /* notify heater that ingredients changed */
        pthread_cond_signal(&readyToBake);
    }
    /* unreachable */
    pthread_mutex_unlock(&lock);
    return NULL;
}

/* egg breaker: each thread adds one egg when bowl needs ingredients */
void* eggBreaker(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        while (numEggInBowl >= 2 || readyToEat) {
            pthread_cond_wait(&needIngredients, &lock);
        }
        addEgg();
        pthread_cond_signal(&readyToBake);
    }
    /* unreachable */
    pthread_mutex_unlock(&lock);
    return NULL;
}

/* bowl heater: waits until bowl has 1 batter and 2 eggs, then heats */
void* bowlHeater(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        while (numBatterInBowl < 1 || numEggInBowl < 2 || readyToEat) {
            pthread_cond_wait(&readyToBake, &lock);
        }
        heatBowl();
        /* wake eater */
        pthread_cond_signal(&startEating);
    }
    /* unreachable */
    pthread_mutex_unlock(&lock);
    return NULL;
}

/* cake eater: waits until cake ready, eats it, resets bowl and notifies adders
     */
void* cakeEater(void* arg) {
    pthread_mutex_lock(&lock);
    while (1) {
        while (!readyToEat) {
            pthread_cond_wait(&startEating, &lock);
        }
        eatCake();
        /* reset bowl for next cake */
        readyToEat = false;
        numBatterInBowl = 0;
        numEggInBowl = 0;
        /* let adders proceed */
        pthread_cond_broadcast(&needIngredients);
    }
    /* unreachable */
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t batter, egg1, egg2, heater, eater;

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&needIngredients, NULL);
    pthread_cond_init(&readyToBake, NULL);
    pthread_cond_init(&startEating, NULL);

    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);

    /* run forever (threads are infinite loops) */
    while (1) sleep(1);

    /* cleanup (unreachable) */
    pthread_join(batter, NULL);
    pthread_join(egg1, NULL);
    pthread_join(egg2, NULL);
    pthread_join(heater, NULL);
    pthread_join(eater, NULL);
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&needIngredients);
    pthread_cond_destroy(&readyToBake);
    pthread_cond_destroy(&startEating);
    return 0;
}
```

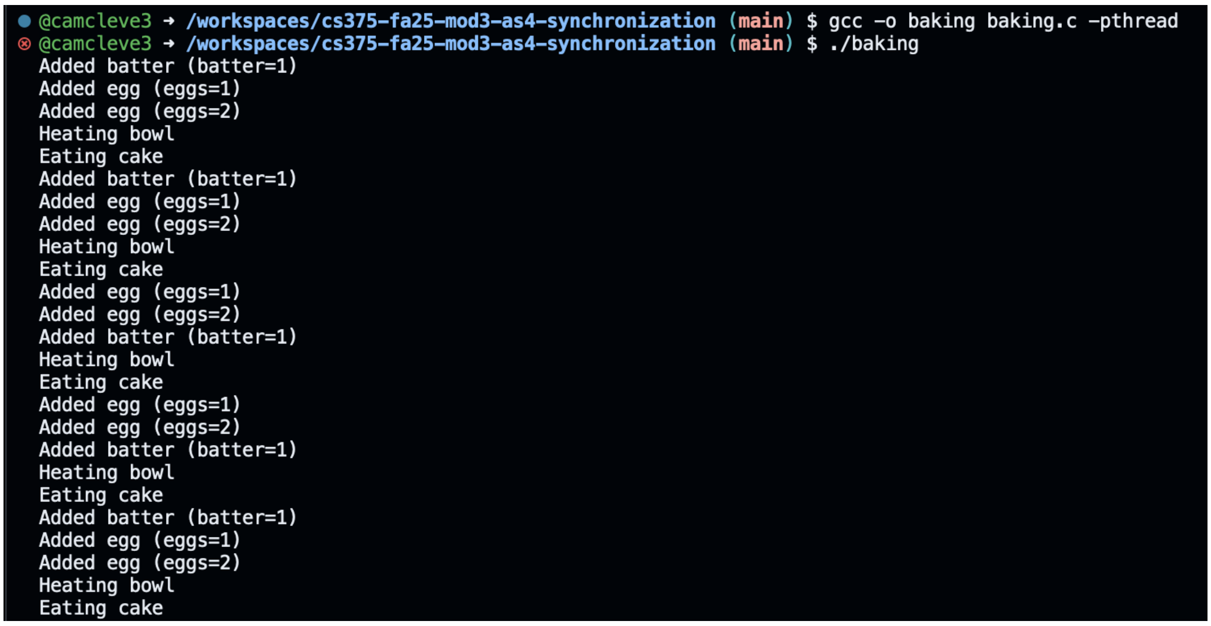
Explanation: There were multiple threads for ingredients, heating, and eating, all needing coordination. Condition variables let each thread wait for the right bowl state: ingredients wait for space, the heater waits for the mix to be ready, and the eater waits until baking is done.

Analysis: This exercise made me think of synchronization as choreography, each step depending on the last one finishing. The mutex protects shared state, and condition variables control timing. Waiting in loops keeps everyone in sync even if timing shifts slightly.

Screenshot:

7 Exercise 6: Priority Donation in Transfer

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct account_t {
    pthread_mutex_t lock;
    int balance;
    long uuid;
```



```
● @camcleve3 ➜ /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o baking baking.c -pthread
⊗ @camcleve3 ➜ /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./baking
  Added batter (batter=1)
  Added egg (eggs=1)
  Added egg (eggs=2)
  Heating bowl
  Eating cake
  Added batter (batter=1)
  Added egg (eggs=1)
  Added egg (eggs=2)
  Heating bowl
  Eating cake
  Added egg (eggs=1)
  Added egg (eggs=2)
  Added batter (batter=1)
  Heating bowl
  Eating cake
  Added egg (eggs=1)
  Added egg (eggs=2)
  Added batter (batter=1)
  Heating bowl
  Eating cake
  Added batter (batter=1)
  Added egg (eggs=1)
  Added egg (eggs=2)
  Heating bowl
  Eating cake
```

Figure 5: Compilation and execution of baking.c

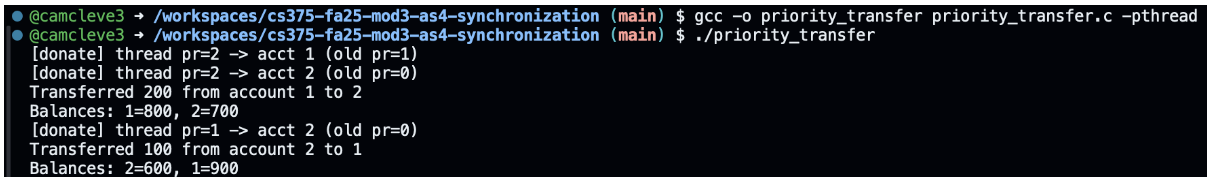
```
    int priority; /* simulated current priority for this account (for donation)
         */
} account_t;

typedef struct transfer_args {
    account_t *donor;
    account_t *recipient;
    int amount;
    int thread_priority;
} transfer_args;

/* transfer with consistent lock ordering and simulated priority donation */
void transfer(account_t *donor, account_t *recipient, int amount, int
    thread_priority) {
    account_t *first = donor;
    account_t *second = recipient;

    /* order locks consistently to avoid deadlock */
    if (donor->uuid > recipient->uuid ||
        (donor->uuid == recipient->uuid && donor > recipient)) {
        first = recipient;
        second = donor;
    }

    /* lock first account */
    pthread_mutex_lock(&first->lock);
    /* simulate donation to first if requestor has higher priority */
    int saved_first_pr = first->priority;
    if (first->priority < thread_priority) {
        printf("[donate]␣thread␣pr=%d␣->␣acct␣%ld␣(old␣pr=%d)\n",
               thread_priority, first->uuid, first->priority);
        first->priority = thread_priority;
    }

    /* lock second account */
    pthread_mutex_lock(&second->lock);
    int saved_second_pr = second->priority;
    if (second->priority < thread_priority) {
```

```c
        printf("[donate] thread pr=%d -> acct %ld (old pr=%d)\n",
               thread_priority, second->uuid, second->priority);
        second->priority = thread_priority;
    }

    /* perform transfer */
    if (donor->balance < amount) {
        printf("Insufficient funds: account %ld has %d, tried to send %d\n",
               donor->uuid, donor->balance, amount);
    } else {
        donor->balance -= amount;
        recipient->balance += amount;
        printf("Transferred %d from account %ld to %ld\n",
               amount, donor->uuid, recipient->uuid);
        printf("Balances: %ld=%d, %ld=%d\n",
               donor->uuid, donor->balance, recipient->uuid, recipient->balance
                  );
    }

    /* restore simulated priorities */
    second->priority = saved_second_pr;
    first->priority = saved_first_pr;

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

void* transfer_thread(void *arg) {
    transfer_args *p = (transfer_args*)arg;
    transfer(p->donor, p->recipient, p->amount, p->thread_priority);
    return NULL;
}

int main(void) {
    account_t acc1, acc2;
    pthread_t t1, t2;
    transfer_args a1, a2;

    acc1.balance = 1000;
    acc1.uuid = 1;
    acc1.priority = 1; /* low */
    pthread_mutex_init(&acc1.lock, NULL);

    acc2.balance = 500;
    acc2.uuid = 2;
    acc2.priority = 0; /* lower */
    pthread_mutex_init(&acc2.lock, NULL);

    /* simulate: thread 1 has high priority (2) transferring from acc1->acc2,
       thread 2 has low priority (1) transferring acc2->acc1 */
    a1.donor = &acc1; a1.recipient = &acc2; a1.amount = 200; a1.thread_priority
         = 2;
    a2.donor = &acc2; a2.recipient = &acc1; a2.amount = 100; a2.thread_priority
         = 1;

    pthread_create(&t1, NULL, transfer_thread, &a1);
    pthread_create(&t2, NULL, transfer_thread, &a2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&acc1.lock);
    pthread_mutex_destroy(&acc2.lock);
    return 0;
}
```

Explanation: Priority inversion happens when a low-priority thread holds a lock that a high-priority thread needs. Priority donation fixes that by temporarily boosting the lock-holder's priority until it finishes the critical section, then lowering it back.

Analysis: This made me appreciate how scheduling and synchronization overlap. Donation prevents high-priority threads from getting stuck behind slower ones, which keeps the system fair and responsive.

Screenshot:



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o priority_transfer priority_transfer.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./priority_transfer
  [donate] thread pr=2 -> acct 1 (old pr=1)
  [donate] thread pr=2 -> acct 2 (old pr=0)
  Transferred 200 from account 1 to 2
  Balances: 1=800, 2=700
  [donate] thread pr=1 -> acct 2 (old pr=0)
  Transferred 100 from account 2 to 1
  Balances: 2=600, 1=900
```

Figure 6: Compilation and execution of priority_transfer.c

8 Exercise 7: Barrier Synchronization

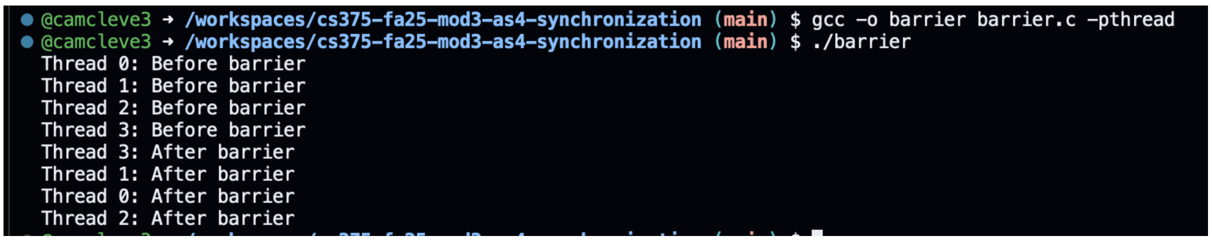
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 4

pthread_mutex_t lock;
pthread_cond_t cv;
int count = 0;
int generation = 0;

void barrier(void) {
    pthread_mutex_lock(&lock);
    int gen = generation;

    count++;
    if (count == NUM_THREADS) {
        /* last thread to arrive: advance generation and wake everyone */
        count = 0;
        generation++;
        pthread_cond_broadcast(&cv);
        pthread_mutex_unlock(&lock);
        return;
    }

    /* wait until generation changes (handles spurious wakeups and reuse) */
    while (gen == generation) {
        pthread_cond_wait(&cv, &lock);
    }
    pthread_mutex_unlock(&lock);
}

void* worker(void* arg) {
    int id = *(int*)arg;
    printf("Thread␣%d:␣Before␣barrier\n", id);
    barrier();
    printf("Thread␣%d:␣After␣barrier\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;
        if (pthread_create(&threads[i], NULL, worker, &ids[i]) != 0) {
            perror("pthread_create");
            return 1;
        }
    }

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

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

Explanation: A barrier makes all threads pause until everyone reaches the same point, then they all continue together. It uses a mutex, an arrival counter, and a generation variable so it can be reused safely for multiple rounds.

Analysis: The generation check is the real trick. It ensures that no thread runs ahead. Every "Before" happens before any "After," which keeps the output clean and the barrier reusable.

Screenshot:



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o barrier barrier.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./barrier
  Thread 0: Before barrier
  Thread 1: Before barrier
  Thread 2: Before barrier
  Thread 3: Before barrier
  Thread 3: After barrier
  Thread 1: After barrier
  Thread 0: After barrier
  Thread 2: After barrier
```

Figure 7: Compilation and execution of barrier.c

9 Exercise 8: Readers-Writers with Priority

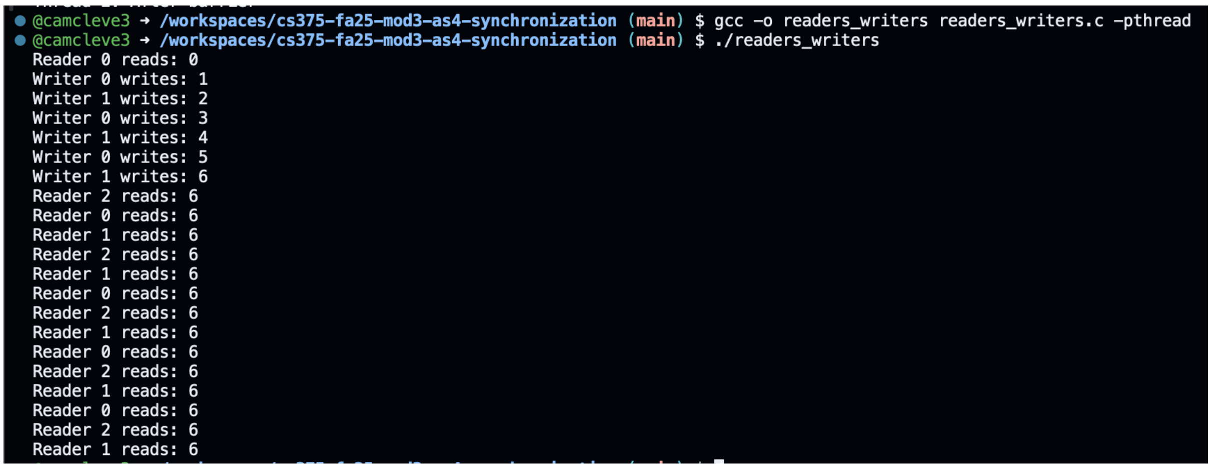
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

pthread_mutex_t lock;
pthread_cond_t reader_cv, writer_cv;
int reader_count = 0;
int writer_waiting = 0;
bool writer_active = false;
int shared_data = 0;

void* reader(void* arg) {
    int id = *(int*)arg;
```

```c
    for (int i = 0; i < 5; ++i) {
        pthread_mutex_lock(&lock);
        while (writer_waiting > 0 || writer_active) {
            pthread_cond_wait(&reader_cv, &lock);
        }
        reader_count++;
        pthread_mutex_unlock(&lock);

        /* perform read */
        printf("Reader␣%d␣reads:␣%d\n", id, shared_data);
        usleep(100000);

        pthread_mutex_lock(&lock);
        reader_count--;
        if (reader_count == 0) {
            pthread_cond_signal(&writer_cv);
        }
        pthread_mutex_unlock(&lock);
        usleep(100000);
    }
    return NULL;
}

void* writer(void* arg) {
    int id = *(int*)arg;
    for (int i = 0; i < 3; ++i) {
        pthread_mutex_lock(&lock);
        writer_waiting++;
        while (reader_count > 0 || writer_active) {
            pthread_cond_wait(&writer_cv, &lock);
        }
        writer_waiting--;
        writer_active = true;
        pthread_mutex_unlock(&lock);

        /* perform write */
        shared_data += 1;
        printf("Writer␣%d␣writes:␣%d\n", id, shared_data);
        usleep(200000);

        pthread_mutex_lock(&lock);
        writer_active = false;
        /* prefer writers: wake one writer first, otherwise wake all readers */
        if (writer_waiting > 0) {
            pthread_cond_signal(&writer_cv);
        } else {
            pthread_cond_broadcast(&reader_cv);
        }
        pthread_mutex_unlock(&lock);
        usleep(100000);
    }
    return NULL;
}

int main(void) {
    const int NREADERS = 3;
    const int NWRITERS = 2;
    pthread_t rthreads[NREADERS], wthreads[NWRITERS];
    int rids[NREADERS], wids[NWRITERS];

    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&reader_cv, NULL);
    pthread_cond_init(&writer_cv, NULL);
    for (int i = 0; i < NREADERS; ++i) {
        rids[i] = i;
        pthread_create(&rthreads[i], NULL, reader, &rids[i]);
    }
    for (int i = 0; i < NWRITERS; ++i) {
        wids[i] = i;
        pthread_create(&wthreads[i], NULL, writer, &wids[i]);
    }

    for (int i = 0; i < NREADERS; ++i) pthread_join(rthreads[i], NULL);
    for (int i = 0; i < NWRITERS; ++i) pthread_join(wthreads[i], NULL);

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

Explanation: Writer priority means if a writer wants access, readers should hold off until the writer is done. The fix blocks new readers while a writer is waiting or active, lets current readers finish, and then wakes the writer.

Analysis: It's a balancing act: give writers priority without starving readers. The mutex guards the shared counters, and condition variables manage entry. It showed me how fair scheduling can be built directly into synchronization logic.

Screenshot:



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o readers_writers readers_writers.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./readers_writers
  Reader 0 reads: 0
  Writer 0 writes: 1
  Writer 1 writes: 2
  Writer 0 writes: 3
  Writer 1 writes: 4
  Writer 0 writes: 5
  Writer 1 writes: 6
  Reader 2 reads: 6
  Reader 0 reads: 6
  Reader 1 reads: 6
  Reader 2 reads: 6
  Reader 1 reads: 6
  Reader 0 reads: 6
  Reader 2 reads: 6
  Reader 1 reads: 6
  Reader 0 reads: 6
  Reader 2 reads: 6
  Reader 1 reads: 6
  Reader 0 reads: 6
  Reader 2 reads: 6
  Reader 1 reads: 6
```

Figure 8: Compilation and execution of readers_writers.c

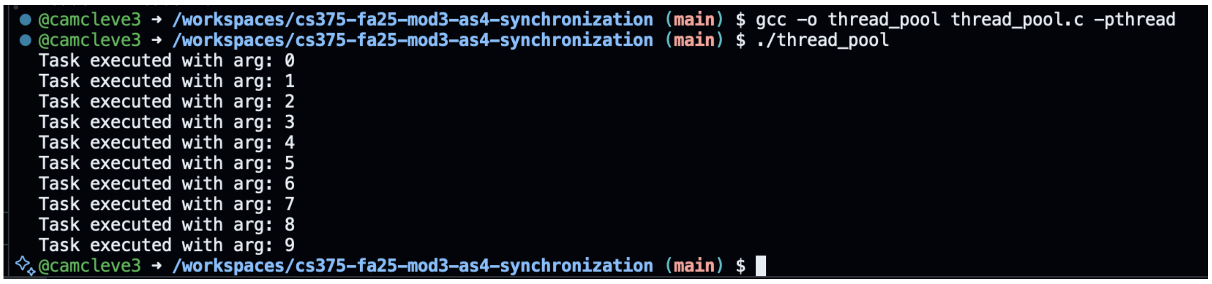
10 Exercise 9: Thread Pool

```
#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;
    pthread_mutex_t lock;
    pthread_cond_t not_empty;
    pthread_cond_t not_full;
} ThreadSafeQueue;

void queue_init(ThreadSafeQueue *q, int size) {
    q->queue = malloc(sizeof(Task) * size);
    if (!q->queue) {
        perror("malloc");
        exit(1);
    }
    q->head = q->tail = q->count = 0;
    q->size = size;
    pthread_mutex_init(&q->lock, NULL);
    pthread_cond_init(&q->not_empty, NULL);
    pthread_cond_init(&q->not_full, NULL);
}

void queue_push(ThreadSafeQueue *q, Task task) {
    pthread_mutex_lock(&q->lock);
    while (q->count == q->size) {
        pthread_cond_wait(&q->not_full, &q->lock);
    }
    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);
}

/* pop returns 1 on success, 0 if queue empty */
int queue_pop(ThreadSafeQueue *q, Task *task) {
    pthread_mutex_lock(&q->lock);
    if (q->count == 0) {
        pthread_mutex_unlock(&q->lock);
        return 0;
    }
    *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;
}

void* worker(void *arg) {
    ThreadSafeQueue *q = (ThreadSafeQueue*)arg;
    while (1) {
        Task t;
        if (queue_pop(q, &t)) {
            t.task(t.arg);
        } else {
            /* no task; back off briefly to avoid busy loop */
            sleep(1);
        }
    }
    return NULL;
}

void sample_task(int arg) {
    printf("Task␣executed␣with␣arg:␣%d\n", arg);
    fflush(stdout);
    usleep(100000); /* simulate work */
}

int main(void) {
    ThreadSafeQueue q;
    queue_init(&q, 10);

    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_create(&threads[i], NULL, worker, &q) != 0) {
            perror("pthread_create");
            return 1;
        }
    }

    for (int i = 0; i < 10; ++i) {
        Task t = { .task = sample_task, .arg = i };
        queue_push(&q, t);
    }

    /* allow tasks to execute, then exit */
    sleep(5);

    /* not strictly necessary in this simple demo, but cleanup would go here */
    return 0;
}
```

Explanation: The thread pool keeps a fixed set of worker threads pulling jobs from a shared queue. The queue uses mutexes and condition variables so producers wait if it's full and workers sleep if it's empty.

Analysis: This design saves a ton of overhead since threads are reused instead of constantly created and destroyed. The synchronization keeps everything safe and efficient, no spinning or wasted CPU time.

Screenshot:



```
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ gcc -o thread_pool thread_pool.c -pthread
● @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ ./thread_pool
  Task executed with arg: 0
  Task executed with arg: 1
  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
  @camcleve3 → /workspaces/cs375-fa25-mod3-as4-synchronization (main) $ 
```

Figure 9: Compilation and execution of thread_pool.c