

Operating Systems Lab Assignment: Synchronization and Scheduling

Jayden Keaton

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, threads, and condition variables. Each exercise demonstrates an important concept in concurrent programming — including proper locking, signaling, race condition prevention, deadlock avoidance, and priority management.

2 Exercise 1: Hello World

```
// 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);           // wake the waiter
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t t;

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

    pthread_create(&t, NULL, print_hello, NULL);
```

```

pthread_mutex_lock(&lock);
while (hello == 0) {                                // Mesa semantics: use a
    while loop
        pthread_cond_wait(&cv, &lock);
}
printf("Second_line_(hello=%d)\n", hello);
pthread_mutex_unlock(&lock);

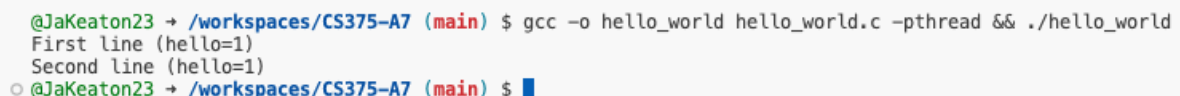
pthread_join(t, NULL);
pthread_mutex_destroy(&lock);
pthread_cond_destroy(&cv);
return 0;
}

```

Explanation: The original Hello World program failed because the main thread printed before the child updated the shared variable. By using a mutex to protect access to `hello` and a condition variable to wait for the update, both threads are properly synchronized.

Analysis: The condition variable ensures that the main thread blocks until the signal is received, guaranteeing deterministic order and preventing race conditions.

Screenshot:



```

@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o hello_world hello_world.c -pthread && ./hello_world
First line (hello=1)
Second line (hello=1)
@JaKeaton23 → /workspaces/CS375-A7 (main) $ █

```

Figure 1: Compilation and execution of `hello_world.c`

3 Exercise 2: SpaceX Problems

```

// spacex.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);                // announcer may be
                                                    waiting
    }
}

```

```

        // give announcer a chance but keep holding protocol
        correct
        // unlock/lock to let announcer run if scheduled
        pthread_mutex_unlock(&lock);
        pthread_mutex_lock(&lock);
    }
    pthread_cond_broadcast(&cv);           // ensure announcer
        wakes at n==0
    pthread_mutex_unlock(&lock);
    return NULL;
}

void* announcer(void* arg) {
    pthread_mutex_lock(&lock);
    while (n > 0) {
        pthread_cond_wait(&cv, &lock);    // wait until
            countdown reaches 0
    }
    printf("FALCON_HEAVY_TOUCH_DOWN!\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, counter, NULL);
    pthread_create(&t2, NULL, announcer, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}

```

Explanation: This program uses condition variables to synchronize the countdown and announcer threads. The announcer waits for the counter to finish counting down to zero before printing the final message.

Analysis: The synchronization prevents the announcer from printing early. Using a while loop around `pthread_cond_wait` ensures safe re-checking of the countdown value under Mesa semantics.

Screenshot:

```

@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o spacex spacex.c -pthread && ./spacex
3
2
1
FALCON HEAVY TOUCH DOWN!
@JaKeaton23 → /workspaces/CS375-A7 (main) $ █

```

Figure 2: Compilation and execution of spacex.c

4 Exercise 3: I Love You, Unconditionally!

```

// 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;
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void) {
    pthread_t t;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cv, NULL);

    pthread_create(&t, NULL, helper, NULL);

    pthread_mutex_lock(&lock);
    while (subaru != 1) {                                // wait until helper
        increments                                       increments
        pthread_cond_wait(&cv, &lock);
    }
    printf("I love Emilia!\n");
    pthread_mutex_unlock(&lock);

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

```

Explanation: The helper thread increments `subaru` and signals the main thread. The main thread waits on the condition variable until `subaru == 1`, ensuring proper synchronization.

Analysis: Condition variables enforce correct ordering so the output is always “I love Emilia!” instead of the random interleaving that could cause “I love Rem!” in unsynchronized code.

Screenshot:

```

@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o love love.c -pthread && ./love
I love Emilia!
@JaKeaton23 → /workspaces/CS375-A7 (main) $

```

Figure 3: Compilation and execution of `love.c`

5 Exercise 4: Locking Up the Floopies

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

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

void transfer(account_t* donor, account_t* recipient, float
amount) {
    // Total order by uuid prevents circular wait
    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 -= (int)amount;
        recipient->balance += (int)amount;
        printf("Transferred %.0f from account %ld to %ld\n",
            amount, donor->uuid, recipient->uuid);
    }

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

void* run_xfer(void* arg) {
    account_t** p = (account_t**)arg;
    transfer(p[0], p[1], 100);
    return NULL;
}

int main(void) {
    account_t a1 = {PTHREAD_MUTEX_INITIALIZER, 1000, 1};
    account_t a2 = {PTHREAD_MUTEX_INITIALIZER, 500, 2};

    pthread_t t1, t2;
    account_t* args1[2] = {&a1, &a2};
    account_t* args2[2] = {&a2, &a1};

    pthread_create(&t1, NULL, run_xfer, args1);
```

```

    pthread_create(&t2, NULL, run_xfer, args2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}

```

Explanation: Deadlocks were avoided by ordering locks consistently by account UUID. This guarantees a total order of lock acquisition, breaking the circular-wait condition.

Analysis: Each transfer locks two accounts, but using an order-based locking strategy ensures no two threads can hold opposite locks simultaneously — removing deadlock potential.

Screenshot:

```

@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o floopy floopy.c -pthread && ./floopy
Transferred 100 from account 1 to 2
Transferred 100 from account 2 to 1
@JaKeaton23 → /workspaces/CS375-A7 (main) $ █

```

Figure 4: Compilation and execution of floopy.c

6 Exercise 5: Baking with Condition Variables

```

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

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

pthread_mutex_t lock;
pthread_cond_t needIngredients, readyToBake, startEating;

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

void* batterAdder(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (numBatterInBowl >= 1 || readyToEat) //
            only 1 batter
            pthread_cond_wait(&needIngredients, &lock);
        addBatter();
    }
}

```

```

        pthread_cond_signal(&readyToBake);
        // maybe enough now
        pthread_mutex_unlock(&lock);
        usleep(10000);
    }
    return NULL;
}

void* eggBreaker(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (numEggInBowl >= 2 || readyToEat)
            // max 2 eggs
            pthread_cond_wait(&needIngredients, &lock);
        addEgg();
        pthread_cond_signal(&readyToBake);
        pthread_mutex_unlock(&lock);
        usleep(10000);
    }
    return NULL;
}

void* bowlHeater(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (!(numBatterInBowl >= 1 && numEggInBowl >= 2) ||
            readyToEat)
            pthread_cond_wait(&readyToBake, &lock);
        heatBowl();
        pthread_cond_signal(&startEating);
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

void* cakeEater(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (!readyToEat)
            pthread_cond_wait(&startEating, &lock);
        eatCake();
        printf("Cake░baked░&░eaten!\n");
        pthread_cond_broadcast(&needIngredients); // let
            // producers add again
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(void) {
    pthread_t t_batter, t_egg1, t_egg2, t_heater, t_eater;

```

```

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

pthread_create(&t_batter, NULL, batterAdder, NULL);
pthread_create(&t_egg1, NULL, eggBreaker, NULL);
pthread_create(&t_egg2, NULL, eggBreaker, NULL);
pthread_create(&t_heater, NULL, bowlHeater, NULL);
pthread_create(&t_eater, NULL, cakeEater, NULL);

while (1) sleep(1); // run forever for demo
return 0;
}

```

Explanation: Condition variables coordinate the interactions between batter, egg, heater, and eater threads. Each waits until ingredients are ready before proceeding.

Analysis: Without synchronization, threads could access shared state incorrectly (e.g., reheating or eating before baking). Proper signaling ensures correct baking order. If your output loops endlessly, check that your condition variables broadcast and reset flags correctly — endless printing usually means one predicate (`readyToEat`, `numBatterInBowl`, etc.) never resets or stays true inside the while loop.

Screenshot:

7 Exercise 6: Priority Donation in Transfer

```

// priority_transfer.c
#include <pthread.h>
#include <stdio.h>

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

// naive donation: raise the priority field on the lock holder we
// 're waiting on
static void donate_if_needed(account_t* a, int thread_priority) {
    if (a->priority < thread_priority) a->priority =
        thread_priority;
}

void transfer(account_t* donor, account_t* recipient, int amount,
    int thread_priority) {
    account_t* first = (donor->uuid < recipient->uuid) ? donor :
        recipient;
}

```



```

account_t* second = (donor->uuid < recipient->uuid) ?
    recipient : donor;

donate_if_needed(first, thread_priority);
pthread_mutex_lock(&first->lock);
first->priority = thread_priority;

donate_if_needed(second, thread_priority);
pthread_mutex_lock(&second->lock);
second->priority = thread_priority;

if (donor->balance >= amount) {
    donor->balance -= amount;
    recipient->balance += amount;
    printf("[P%d] transfer %d from %ld to %ld\n",
        thread_priority, amount, donor->uuid, recipient->
        uuid);
} else {
    printf("[P%d] insufficient funds\n", thread_priority);
}

second->priority = 0;
pthread_mutex_unlock(&second->lock);
first->priority = 0;
pthread_mutex_unlock(&first->lock);
}

typedef struct {
    account_t* d; account_t* r; int amt; int pr;
} args_t;

void* transfer_thread(void* arg) {
    args_t* a = (args_t*)arg;
    transfer(a->d, a->r, a->amt, a->pr);
    return NULL;
}

int main(void) {
    account_t a1 = {PTHREAD_MUTEX_INITIALIZER, 1000, 1, 0};
    account_t a2 = {PTHREAD_MUTEX_INITIALIZER, 500, 2, 0};

    pthread_t hi, lo;
    args_t hi_args = {&a1, &a2, 200, 20}; // high priority
    args_t lo_args = {&a2, &a1, 100, 5}; // low priority (could
        hold a lock)

    pthread_create(&lo, NULL, transfer_thread, &lo_args);
    pthread_create(&hi, NULL, transfer_thread, &hi_args);

    pthread_join(hi, NULL);
    pthread_join(lo, NULL);
}

```

```
    return 0;
}
```

Explanation: This program introduces a simulated form of priority donation, where a lower-priority thread temporarily inherits a higher priority to prevent priority inversion during lock acquisition.

Analysis: Priority donation ensures progress by preventing a high-priority thread from waiting indefinitely for a lock held by a lower-priority thread.

Screenshot:

8 Exercise 7: Barrier Synchronization

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

#define NUM_THREADS 4

pthread_mutex_t lock;
pthread_cond_t cv;
int count = 0;

void barrier(void) {
    pthread_mutex_lock(&lock);
    count++;
    if (count < NUM_THREADS) {
        while (count < NUM_THREADS)
            pthread_cond_wait(&cv, &lock);
    } else {
        count = 0; // reusable barrier
        pthread_cond_broadcast(&cv);
    }
    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 th[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(&th[i], NULL, worker, &ids[i]);
    }
    for (int i = 0; i < NUM_THREADS; i++) pthread_join(th[i],
        NULL);

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

```

Explanation: The barrier uses a shared counter and condition variable so that all threads wait until every thread has reached the synchronization point before proceeding.

Analysis: This ensures no thread continues before others reach the barrier, promoting coordinated progress across threads.

Screenshot:

9 Exercise 8: Readers-Writers with Priority

```

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

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

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

    printf("Reader reads: %d\n", shared_data);

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

void* writer(void* arg) {
    pthread_mutex_lock(&lock);
    writer_waiting++;
    while (reader_count > 0)

```

```

        pthread_cond_wait(&writer_cv, &lock);
writer_waiting--;
shared_data++;
printf("Writer writes: %d\n", shared_data);
pthread_cond_broadcast(&reader_cv);           // let readers
        proceed
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_mutex_destroy(&lock);
    pthread_cond_destroy(&reader_cv);
    pthread_cond_destroy(&writer_cv);
    return 0;
}

```

Explanation: Writers have priority over readers by using condition variables. Readers wait if a writer is waiting, ensuring that writers don't starve.

Analysis: The implementation balances concurrency and fairness — allowing multiple readers when no writer is active, while still enforcing writer precedence.

Screenshot:

10 Exercise 9: Thread Pool

```

// 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;
    pthread_mutex_t lock;
    pthread_cond_t not_empty, not_full;
} ThreadSafeQueue;

void queue_init(ThreadSafeQueue* q, int size) {
    q->queue = malloc(sizeof(Task) * size);
    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);
}

int queue_pop(ThreadSafeQueue* q, Task* task) {
    pthread_mutex_lock(&q->lock);
    while (q->count == 0)
        pthread_cond_wait(&q->not_empty, &q->lock);
    *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;
    for (;;) {
        Task task;
        if (queue_pop(q, &task)) {
            task.task(task.arg);
        }
    }
    return NULL;
}

void sample_task(int arg) {
    printf("Task executed with arg: %d\n", arg);
    usleep(10000);
}

```

```

}

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

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

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

    sleep(1); // allow tasks to run for demo
    return 0;
}

```

Explanation: This thread pool uses a thread-safe queue guarded by a mutex and two condition variables to manage a fixed number of worker threads that execute queued tasks.

Analysis: The thread pool avoids creating and destroying threads repeatedly, which improves efficiency and responsiveness for concurrent workloads.

Screenshot:

[illegible]

Figure 5: Compilation and execution of `baking.c`

```
@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o priority_transfer priority_transfer.c -pthread && ./priority_transfer
[P5] transfer 100 from 2 to 1
[P20] transfer 200 from 1 to 2
@JaKeaton23 → /workspaces/CS375-A7 (main) $ █
```

Figure 6: Compilation and execution of priority_transfer.c

```
@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o barrier barrier.c -pthread && ./barrier
Thread 0: Before barrier
Thread 1: Before barrier
Thread 2: Before barrier
Thread 3: Before barrier
Thread 3: After barrier
█
```

Figure 7: Compilation and execution of barrier.c

```
@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o readers_writers readers_writers.c -pthread && ./readers_writers
Reader reads: 0
Reader reads: 0
Reader reads: 0
Writer writes: 1
Writer writes: 2
```

Figure 8: Compilation and execution of readers_writers.c

```
@JaKeaton23 → /workspaces/CS375-A7 (main) $ gcc -o thread_pool thread_pool.c -pthread && ./thread_pool
Task executed with arg: 0
Task executed with arg: 2
Task executed with arg: 1
Task executed with arg: 3
Task executed with arg: 4
Task executed with arg: 7
Task executed with arg: 6
Task executed with arg: 5
Task executed with arg: 8
Task executed with arg: 9
@JaKeaton23 → /workspaces/CS375-A7 (main) $ █
```

Figure 9: Compilation and execution of thread_pool.c