

HY345 - Operating Systems

Post Office - 2nd Assignment

Threads, Bounded Buffers, Synchronization

Threads

- A thread is a light-weight process
- A thread exists within a process and uses the process resources
- It is asynchronous
- The program in C calls the pthread.h header file
- **How to compile:** gcc main.c postoffice.c -o postoffice -lpthread

Creating a Thread

Function Signature:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*func)(void*), void *arg);
```

- 1st argument: (*thread) - pointer to the identifier of the created thread
- 2nd argument: (*attr) - thread attributes, if NULL, thread is created with default attributes
- 3rd argument: (*func) - pointer to the function the thread will execute
- 4th argument: (*arg) - the argument passed to the thread function

Creating a Thread (code example)

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

void *my_thread(void *arg) {
    printf("I am a thread!\n");
    return NULL;
}

int main() {
    pthread_t tid;

    pthread_create(&tid, NULL, my_thread, NULL);
    pthread_join(tid, NULL);

    return 0;
}
```

Passing Arguments to a Thread

```
typedef struct {
    int id;
    int age;
} Person;
void *my_thread(void *arg) {
    Person *p = (Person *)arg; // cast void back to struct
    printf("Thread id: %d, age: %d\n", p->id, p->age);
    return NULL;
}
int main() {
    pthread_t tid;
    Person alice = {1, 25};
    pthread_create(&tid, NULL, my_thread, &alice);
    pthread_join(tid, NULL);
}
```

- A thread function can only receive **ONE** argument through **void ***
- Pack everything the thread needs into a **struct**
- Pass the address of the struct to **pthread_create**

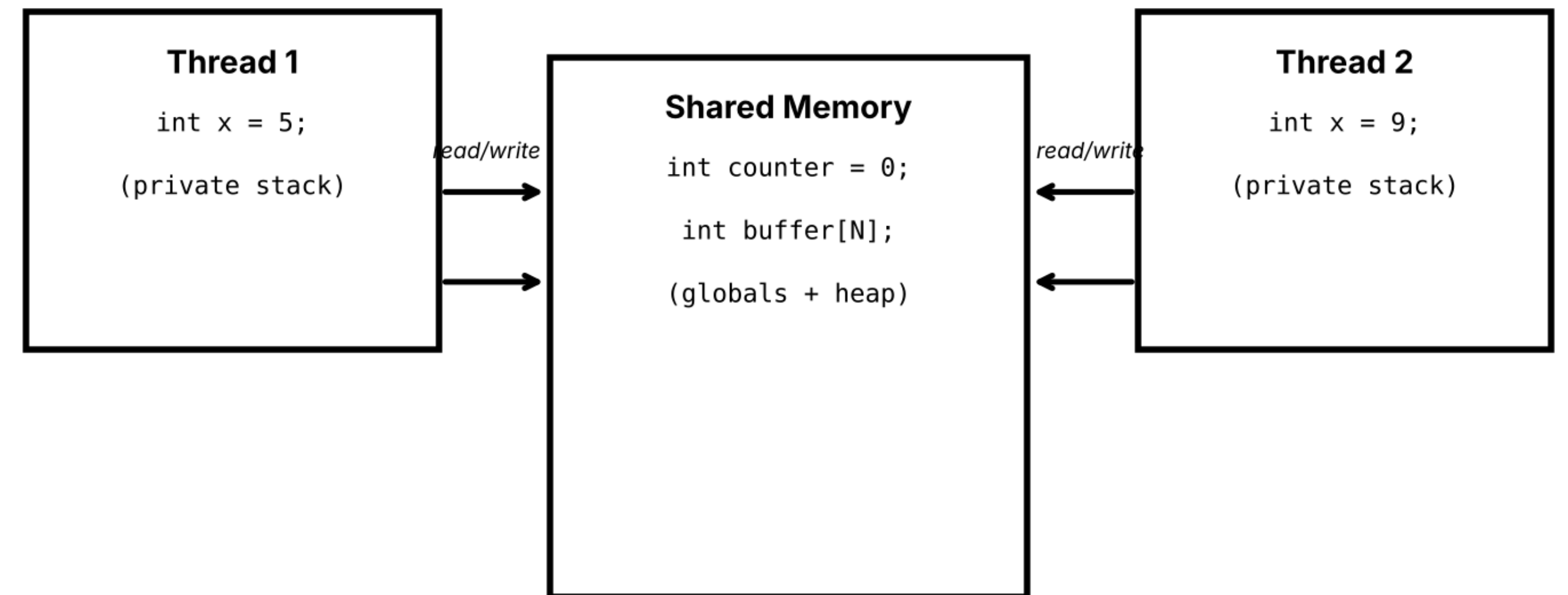
Joining Threads

```
for (int i = 0; i < N; i++)  
    pthread_create(&tids[i], NULL, my_func, &args[i]);  
  
for (int i = 0; i < N; i++)  
    pthread_join(tids[i], NULL);
```

- **pthread_join(tid, NULL)** makes the calling thread wait until **tid** finishes
- Without join, **main()** may exit before threads finish
- **Always** create **ALL** threads first, then join **ALL** threads
- **Never** join inside the creation loop - threads would run one at a time

Shared Memory

- A shared memory segment is a portion of memory shared between multiple threads
- In this assignment we are dealing with intra-process communication
- All global variables and malloc'd memory are shared between threads
- Each thread has its own private stack (local variables)



Example of Two threads sharing global memory

Shared Memory - Concerns

- Needs concurrency control / synchronization → Data inconsistencies are possible
- Two threads updating a shared counter simultaneously without synchronization causes **race conditions**
- Threads must be informed when it is safe to read and write shared data

Thread Synchronization Mechanisms

- Mutex Exclusion (**mutex**)
 - Used to serialize access to shared memory
 - It is a locking mechanism
- Semaphores
 - A counter with blocking behavior
 - Controls access to a quantity (slots, items)
 - It is a signaling mechanism

Mutexes

- **Guard** against multiple threads modifying the same shared data simultaneously
- Provide **locking/unlocking** of critical code sections where shared data is modified
- Each thread **waits for the mutex** to be unlocked before entering the critical section
- A mutex is **unlocked** only **by the thread that locked it**

Mutexes - Create and Initialize

- **Mutex variables** are declared with type **pthread_mutex_t**
- Two ways to initialize:
 Statically: pthread_mutex_t mut= PTHREAD_MUTEX_INITIALIZER
 Dynamically: pthread_mutex_init(&mut, NULL) - use this
 when mutex is inside a struct
- Routines:
 pthread_mutex_init(mutex, attr)
 pthread_mutex_destroy(mutex)

Mutexes - Basic Functions

```
pthread_mutex_lock(pthread_mutex_t *mutex);  
pthread_mutex_unlock(pthread_mutex_t *mutex);  
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- A mutex is like a **key** handed to **only one thread at a time**
- The lock/unlock functions work together
- A mutex is **unlocked** only **by the thread that has locked it**

Mutexes - Code Example

```
pthread_mutex_t lock;

int delivered = 0; // shared variable

void *worker_thread(void *arg) {
    pthread_mutex_lock(&lock);
    counter++;          // critical section
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_mutex_init(&lock, NULL);
    // create threads...
    // join threads...
    pthread_mutex_destroy(&lock);
}
```

When a thread reaches the lock, it first checks if the mutex is taken:

- If free → it takes the lock, does its work, then releases
- If taken → it waits until unlocked

Semaphores

- A semaphore is a counter with blocking behavior
- When the counter reaches 0, any thread trying to decrement it **blocks**
- When a thread increments it, a blocked thread **wakes up**
- Should include the **semaphore.h** header file
- Semaphore functions use **sem_** prefix, not **pthread_**

Two types:

- **Counting semaphore** - tracks a quantity (free slots, available items)
- **Binary semaphore** - starts at 0 or 1, used for signaling between two threads

Semaphores - Basic Functions (1/2)

Creating:

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

- **pshared=0** → shared between threads of the same process
- **value** → initial counter value

Terminating:

```
sem_destroy(sem_t *sem);
```

- Frees resources allocated to the semaphore

Semaphores - Basic Functions (2/2)

Signaling:

`sem_post(sem_t *sem);`

- Atomically increases the semaphore value by 1
- Wakes a sleeping thread if any

Waiting:

`sem_wait(sem_t *sem);`

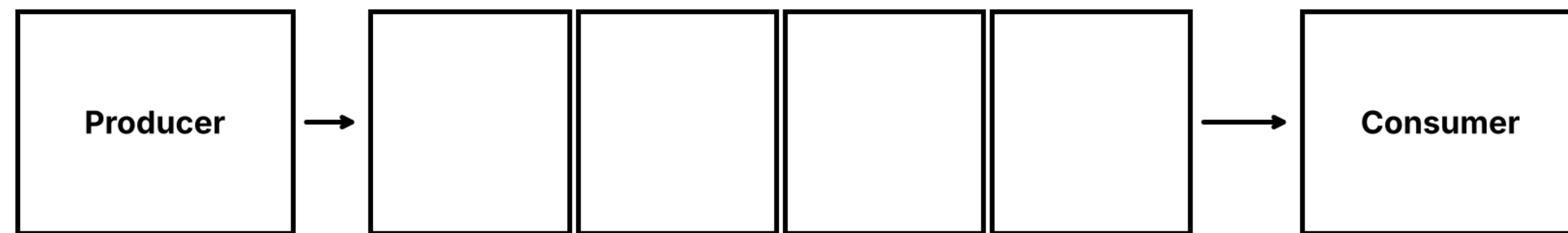
- Atomically decreases the semaphore value by 1
- Always waits until the semaphore has a non-zero value first

Comparison Table

	Semaphore	Mutex
Use case	Signaling mechanism	Locking mechanism
Types	Counting:1,2,3...n or Binary: 0 and 1	No types
Operations	sem_post/sem_wait	lock/unlock
Blocks when	Counter = 0	Lock is taken

The Bounded Buffer

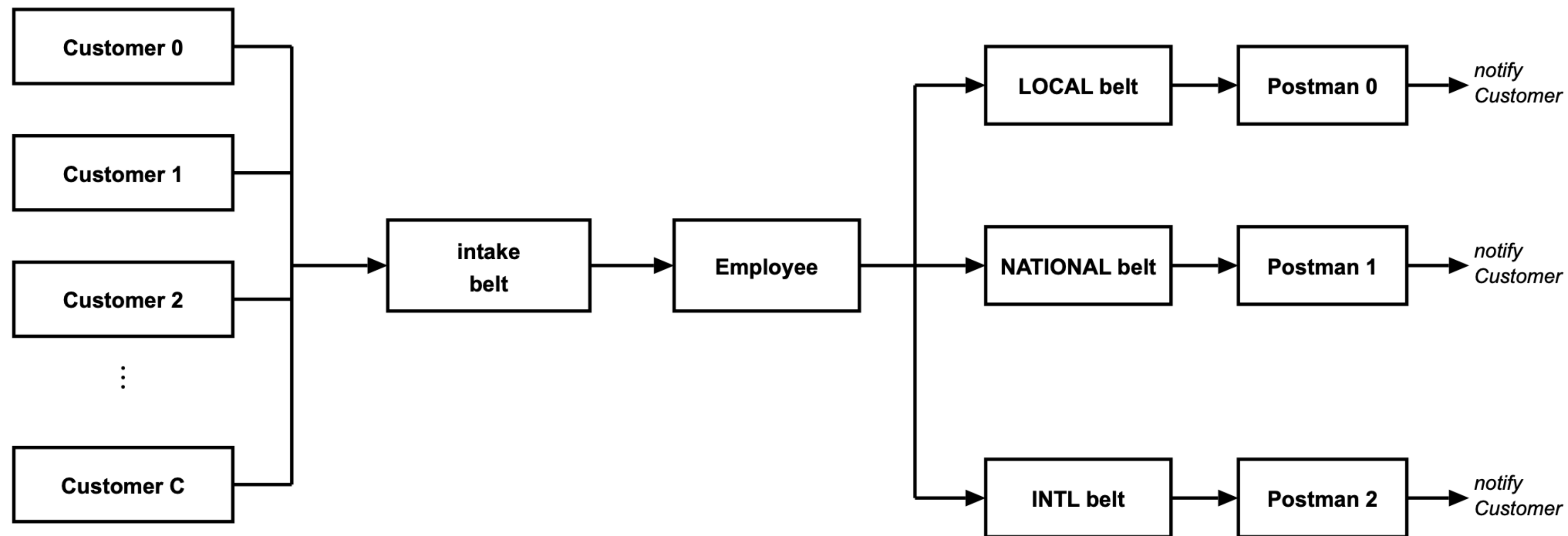
- A bounded buffer is a fixed-size storage between a producer and a consumer
- If the buffer is full → producer waits
- If the buffer is empty → consumer waits
- In this assignment every belt is a bounded buffer



capacity = 4

The Post Office Problem

- Every arrow crossing a belt is a potential race condition
- Every belt is a bounded buffer - needs 2 semaphores + 1 mutex
- Program stops after all C parcels delivered



Thread Types in the Assignment

- **Customer thread: creates parcel, places on intake belt, waits for delivery notification**
- **Employee thread: picks from intake belt, classifies parcel, places on destination belt**
- **Postman thread: picks from destination belt, delivers, notifies customer**
- **Main thread: initializes everything, creates all threads, joins all threads, cleans up**

File Structure and Compilation

Required Files:

postoffice.h → structs and function declarations
postoffice.c → belt functions and thread functions
main.c → argument parsing, thread creation, cleanup
Makefile → build rules
README.md → implementation description

Compile command:

```
gcc main.c postoffice.c -o postoffice -lpthread -Wall
```

Run command:

```
./postoffice <num_customers> <num_employees> <belt_capacity>
```

Example 1: Hello World from a thread

```
#include <stdio.h>
#include <pthread.h>
// Thread function: must take void* and return void*
void *hello(void *arg) {
    printf("Hello World from thread!\n");
    return NULL;
}
int main() {
    pthread_t t;
    pthread_create(&t, NULL, hello, NULL); //create thread
    pthread_join(t, NULL); //wait for it to finish
    printf("Back in main\n");
    return 0;
}
```

Example 2: Two threads, race condition → then fix with mutex (1/2)

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

int counter=0; // shared variable - no protection

void *increment(void *arg){
    for(int i=0;i<100000; i++)
        counter++; // read → add 1 → write (3 steps, not atomic!)
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d (expected 200000)\n", counter);
    //You will almost never see 200000 - race condition!
    return 0;
}
```

BROKEN!

Example 2: Two threads, race condition → then fix with mutex (2/2)

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

int counter=0;
pthread_mutex_t lock= PTHREAD_MUTEX_INITIALIZER; //the fix

void *increment(void *arg){
    for(int i=0;i<100000;i++){
        pthread_mutex_lock(&lock); //only one thread enters here at a time
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}

int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Counter = %d (expected 200000)\n", counter);
    //Now always 200000
    return 0;
}
```

FIXED!

Thank you for your time!

Questions?