

H10-1

Source code (pc.c)

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

#define BUF_SIZE 4    // buffer size (bounded buffer)
#define LOOPS 20     // how many items to produce/consume

int buffer[BUF_SIZE];
int fill = 0;        // write index
int use = 0;         // read index

sem_t empty;         // number of empty slots
sem_t full;           // number of full slots
sem_t mutex;         // binary semaphore for mutual exclusion

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % BUF_SIZE;
}

int get(void) {
    int tmp = buffer[use];
    use = (use + 1) % BUF_SIZE;
    return tmp;
}

void *producer(void *arg)
{
    int i;
    for (i = 0; i < LOOPS; i++) {
        sem_wait(&empty); // wait until buffer has empty slot
        sem_wait(&mutex); // enter critical section

        put(i);           // access shared buffer

        sem_post(&mutex); // leave critical section
        sem_post(&full);  // one more full slot
    }
    return NULL;
}

void *consumer(void *arg)
{
    int i, tmp;
    for (i = 0; i < LOOPS; i++) {
        sem_wait(&full); // wait until buffer has data
        sem_wait(&mutex); // enter critical section

        tmp = get();     // access shared buffer

        sem_post(&mutex); // leave critical section
        sem_post(&empty); // one more empty slot

        printf("consume %d\n", tmp);
    }
    return NULL;
}

int main(void)
{
    pthread_t prod, cons;

    sem_init(&empty, 0, BUF_SIZE); // start with all slots empty
    sem_init(&full, 0, 0);         // no full slots
    sem_init(&mutex, 0, 1);        // binary lock

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
```

```

pthread_join(prod, NULL);
pthread_join(cons, NULL);

sem_destroy(&empty);
sem_destroy(&full);
sem_destroy(&mutex);

return 0;
}

```

Explanation

In slide 12, empty and full only control how many slots are used, but there is no mutual exclusion: two threads can call put() or get() at the same time and update buffer, fill, or use concurrently → race condition.

In my solution I add a binary semaphore mutex. Both producer and consumer do sem_wait(&mutex) before calling put()/get() and sem_post(&mutex) after. This makes the buffer and index updates a critical section that only one thread can execute at a time.

empty and full still prevent overflow/underflow of the buffer, and mutex prevents concurrent access to shared variables, so the race condition is removed.

```
gcc pc.c -o pc -pthread
```

```
./pc
```

H10-2

1. Explanation: Why User_task0() cannot enter the critical section

In the code of Slide 22, Test_critical_section() uses Kernel_lock_sem() to enter the critical section. User_task2() runs first and successfully locks the semaphore, so the semaphore value becomes 0.

The semaphore is never properly released for User_task0() because the unlock operation is only done by a separate event and not by the task that entered the critical section. As a result, when we press 'X', User_task0() calls Kernel_lock_sem(), but Kernel_sem_test() always returns false (semaphore still 0), so User_task0() is stuck in the loop inside Kernel_lock_sem() and never reaches the critical section.

In short: the semaphore is locked by another task and never released in a way that lets User_task0() acquire it, so User_task0() fails to enter the critical section.

2. Fixed code (with comments)

Below is a clean version that follows the lab PDF and solves the problem by:

Adding a mutex (with owner).

Locking the mutex in Test_critical_section().

Generating an Unlock event from UART.

Handling that event and calling Kernel_unlock_mutex().

You only need to merge these pieces into your existing project.

2.1 synch.h – add mutex type and prototypes

```
// synch.h
#ifndef KERNEL_SYNCH_H_
```

```

#define KERNEL_SYNCH_H_

#include <stdint.h>
#include <stdbool.h>

// ----- semaphore -----
void Kernel_sem_init(int32_t max);
bool Kernel_sem_test(void);
void Kernel_sem_release(void);

// ----- mutex (NEW) -----
typedef struct KernelMutex_t {
    uint32_t owner; // task ID that owns the mutex
    bool lock; // true if locked
} KernelMutex_t;

void Kernel_mutex_init(void);
bool Kernel_mutex_lock(uint32_t owner);
bool Kernel_mutex_unlock(uint32_t owner);

#endif /* KERNEL_SYNCH_H_ */

```

2.2 synch.c – implement mutex

```

// synch.c
#include "synch.h"

#define DEF_SEM_MAX 8

static int32_t sSemMax;
static int32_t sSem;
static KernelMutex_t sMutex;

// ----- semaphore code (already in lab) -----

void Kernel_sem_init(int32_t max) {
    sSemMax = (max <= 0) ? DEF_SEM_MAX : max;
    sSemMax = (max >= DEF_SEM_MAX) ? DEF_SEM_MAX : max;
    sSem = sSemMax;
}

bool Kernel_sem_test(void) {
    if (sSem <= 0) return false;
    sSem--;
    return true;
}

void Kernel_sem_release(void) {
    sSem++;
    if (sSem >= sSemMax) sSem = sSemMax;
}

// ----- mutex implementation (FIX) -----

void Kernel_mutex_init(void) {
    sMutex.owner = 0;
    sMutex.lock = false;
}

bool Kernel_mutex_lock(uint32_t owner) {
    if (sMutex.lock) {
        // someone else already owns the mutex
        return false;
    }
    sMutex.owner = owner;
    sMutex.lock = true;
}

```

```

    return true;
}

bool Kernel_mutex_unlock(uint32_t owner){
    // only the task that locked the mutex can unlock it
    if (owner == sMutex.owner) {
        sMutex.lock = false;
        return true;
    }
    return false;
}

```

2.3 Kernel.h – add mutex API

```

// Kernel.h
...
void Kernel_lock_sem(void);
void Kernel_unlock_sem(void);

// NEW: mutex APIs
void Kernel_lock_mutex(void);
void Kernel_unlock_mutex(void);
...

```

2.4 Kernel.c – implement lock / unlock for mutex

```

// Kernel.c
#include "Kernel.h"
#include "synch.h"
#include "task.h" // for Kernel_task_get_current_task_id()

// ---- semaphore API (given in lab) ----
void Kernel_lock_sem(void) {
    while (false == Kernel_sem_test()) {
        Kernel_yield();
    }
}

void Kernel_unlock_sem(void) {
    Kernel_sem_release();
}

// ---- mutex API (FIX) ----
void Kernel_lock_mutex(void) {
    while (true) {
        uint32_t current_task_id = Kernel_task_get_current_task_id();

        // try to lock; if fail, yield and try again
        if (false == Kernel_mutex_lock(current_task_id)) {
            Kernel_yield();
        } else {
            break; // locked successfully
        }
    }
}

void Kernel_unlock_mutex(void) {
    uint32_t current_task_id = Kernel_task_get_current_task_id();

    // only owner can successfully unlock; otherwise just yield
    if (false == Kernel_mutex_unlock(current_task_id)) {
        Kernel_yield();
    }
}

```

2.5 Main.c – initialize mutex and fix Test_critical_section()

```

// Main.c

```

```

static uint32_t shared_value;

// Called at boot
static void Kernel_init(void) {
    uint32_t taskId;

    Kernel_task_init();
    Kernel_event_flag_init();
    Kernel_mutex_init(); // FIX: initialize mutex

    ...
}

// Function that uses the critical section
static void Test_critical_section(uint32_t p, uint32_t taskId)
{
    debug_printf("User Task #%u Try to access critical section\n", taskId);

    Kernel_lock_mutex(); // FIX: protect critical section with mutex

    debug_printf("User Task #%u Send=%u\n", taskId, p);
    shared_value = p;
    Kernel_yield();
    delay(1000);
    debug_printf("User Task #%u Shared Value=%u\n", taskId, shared_value);

    Kernel_unlock_mutex(); // FIX: release mutex when done
}

User_task0() stays the same, it just calls Test_critical_section(5, 0);

```

2.6 event.h – add Unlock event flag

```

// event.h
typedef enum KernelEventFlag_t {
    KernelEventFlag_UartIn = 0x00000001,
    KernelEventFlag_CmdIn = 0x00000002,
    KernelEventFlag_CmdOut = 0x00000004,

    KernelEventFlag_Unlock = 0x00000008, // FIX: event to unlock mutex

    KernelEventFlag_Empty = 0x00000000,
} KernelEventFlag_t;

```

2.7 Uart.c – generate unlock event on key press

```

// Uart.c

static void interrupt_handler(void)
{
    uint8_t ch = Hal_uart_get_char();

    if (ch == 'U') {
        // FIX: pressing 'U' generates an Unlock event
        Kernel_send_events(KernelEventFlag_Unlock);
    }

    if (ch != 'X') {
        // Normal characters: echo and send to Task0
        Hal_uart_put_char(ch);
        Kernel_send_msg(KernelMsgQ_Task0, &ch, 1);
        Kernel_send_events(KernelEventFlag_UartIn);
    } else {
        // 'X' requests User_task0() to try the critical section
        Kernel_send_events(KernelEventFlag_CmdOut);
    }
}

```

```
}
```

2.8 User_task1() – handle unlock event

// Main.c (user task part)

```
void User_task1(void)
{
    uint32_t local = 0;
    debug_printf("User Task #1 SP=0x%x\n", &local);

    uint8_t cmdlen = 0;
    uint8_t cmd[16] = {0};

    while (true) {
        // FIX: wait for either command input or unlock event
        KernelEventFlag_t handle_event =
            Kernel_wait_events(KernelEventFlag_CmdIn | KernelEventFlag_Unlock);

        switch (handle_event) {
            case KernelEventFlag_CmdIn:
                memclr(cmd, 16);
                Kernel_recv_msg(KernelMsgQ_Task1, &cmdlen, 1);
                Kernel_recv_msg(KernelMsgQ_Task1, cmd, cmdlen);
                debug_printf("\nRecv Cmd: %s\n", cmd);
                break;

            case KernelEventFlag_Unlock:
                // FIX: Unlock the mutex so other tasks (e.g. User_task0)
                // can enter the critical section
                Kernel_unlock_mutex();
                break;
        }

        Kernel_yield();
    }
}
```

H10-3

1. Why does User_task2() only enter the critical section once?

After we switch from **semaphore** to **mutex**, the mutex implementation is:

```
bool Kernel_mutex_unlock(uint32_t owner)
{
    if (owner == sMutex.owner) { // only owner can unlock
        sMutex.lock = false;
        return true;
    }
    return false;
}
```

and Kernel_unlock_mutex() does:

```
void Kernel_unlock_mutex(void)
{
    uint32_t current_task_id = Kernel_task_get_current_task_id();
    if (false == Kernel_mutex_unlock(current_task_id)) {
        Kernel_yield();
    }
}
```

In Slide 25:

- User_task2() locks the mutex inside Test_critical_section() and becomes **owner**.
- When we press **U**, the UART ISR sends KernelEventFlag_Unlock.
- **User_task1()** is the one that wakes up on that event and calls Kernel_unlock_mutex().

But Kernel_unlock_mutex() uses **the current task id** (User_task1), while the mutex owner is **User_task2**, so owner != sMutex.owner → Kernel_mutex_unlock() returns false

The mutex **stays locked**, therefore User_task2() can enter the critical section the first time (when the mutex is initially unlocked), but never again, **even though we keep pressing U**.

Reason in one line (for your report)

After adding the owner check, the mutex is locked by User_task2() but User_task1() tries to unlock it when 'U' is pressed, so the owner check fails and the mutex never becomes unlocked again.

2. Fix: the owner task must unlock the mutex

To solve this properly:

1. **Only the task that locked the mutex** (e.g. User_task2) should call Kernel_unlock_mutex().
2. We still want to use the 'U' key, so we let the *owner* task wait for the Unlock event and unlock itself.
3. User_task1() should no longer handle KernelEventFlag_Unlock.

Below is a ready-made version of the relevant code with comments you can submit.

2.1 event.h – keep the Unlock flag (unchanged)

```
typedef enum KernelEventFlag_t {
    KernelEventFlag_UartIn = 0x00000001,
    KernelEventFlag_CmdIn = 0x00000002,
    KernelEventFlag_CmdOut = 0x00000004,
    KernelEventFlag_Unlock = 0x00000008, // event generated by 'U' key
    KernelEventFlag_Empty = 0x00000000,
} KernelEventFlag_t;
```

2.2 Uart.c – still generate Unlock event on 'U'

```
static void interrupt_handler(void)
{
    uint8_t ch = Hal_uart_get_char();

    if (ch == 'U') {
        // When user presses 'U', send Unlock event
        Kernel_send_events(KernelEventFlag_Unlock);
    }

    if (ch != 'X') {
        Hal_uart_put_char(ch);
        Kernel_send_msg(KernelMsgQ_Task0, &ch, 1);
        Kernel_send_events(KernelEventFlag_UartIn);
    } else {
        // 'X' asks User_task0() to try the critical section
        Kernel_send_events(KernelEventFlag_CmdOut);
    }
}
```

2.3 User_task1() – remove Unlock handling

```
void User_task1(void)
{
    uint32_t local = 0;
    debug_printf("User Task #1 SP=0x%x\n", &local);

    uint8_t cmdlen = 0;
    uint8_t cmd[16] = {0};

    while (1) {
        // FIX: wait only for command input; no longer for Unlock
        KernelEventFlag_t handle_event =
            Kernel_wait_events(KernelEventFlag_CmdIn);

        switch (handle_event) {
            case KernelEventFlag_CmdIn:
                memclr(cmd, 16);
                Kernel_rcv_msg(KernelMsgQ_Task1, &cmdlen, 1);
                Kernel_rcv_msg(KernelMsgQ_Task1, cmd, cmdlen);
                debug_printf("\nRecv Cmd: %s\n", cmd);
                break;
        }
    }
```

```

    Kernel_yield();
}
}

```

Comment idea:

// FIX: User_task1 must not unlock the mutex because it is not the owner; owner check would fail and the mutex would stay locked.

2.4 Main.c – make the *owner* task wait for Unlock and then unlock

```
static uint32_t shared_value;
```

```

static void Test_critical_section(uint32_t p, uint32_t taskId)
{
    debug_printf("User Task #%%u Try to access critical section\\n", taskId);

    Kernel_lock_mutex(); // taskId becomes the owner of the mutex

    debug_printf("User Task #%%u Send=%%u\\n", taskId, p);
    shared_value = p;
    Kernel_yield();
    delay(1000);
    debug_printf("User Task #%%u Shared Value=%%u\\n", taskId, shared_value);

    // FIX: owner task waits until user presses 'U' (Unlock event)
    debug_printf("User Task #%%u waiting for Unlock event...\\n", taskId);
    Kernel_wait_events(KernelEventFlag_Unlock);

    // FIX: mutex is now unlocked by the same task that locked it
    Kernel_unlock_mutex();
}

```

Comment idea:

// FIX: Unlock must be called by the owner task (User_task2/User_task0). We move the unlock logic here so the owner waits for KernelEventFlag_Unlock and then releases the mutex itself.