Operating System (CSE231)

Section-B

Quiz 3

Total Marks: 40 **Time: 1 hr**

---

1. (15 points) Answer the following questions.

   (a) (8 points) Consider the following program that increments a shared counter using two threads. The programmer wants to ensure that the final counter value is correct. Fill in the blanks in the code.

```
#include <stdio.h>
#include <pthread.h>
#include "common_threads.h"
static int counter = 0;
_______ mylock;                                  // (1) Declare a mutex lock

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    for (int i = 0; i < 1e7; i++) {
        _______( &mylock );                      // (2) Acquire the lock
        counter = counter + 1;
        _______( &mylock );                      // (3) Release the lock
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    _______( &mylock, NULL );                    // (4) Initialize the mutex
    printf("main: begin (counter = %d)\n", counter);
    _______( &p1, NULL, mythread, "A" );         // (5.1) Create thread A
    _______( &p2, NULL, mythread, "B" );         // (5.2) Create thread B
    _______( p1, NULL );                         // (6.1) Wait for thread A
    _______( p2, NULL );                         // (6.2) Wait for thread B
    printf("main: done (counter = %d)\n", counter);
    pthread_mutex_destroy( &mylock );            // (7) Destroy the mutex
    return 0;}
```

Blank	Answer
(1)	pthread_mutex_t
(2)	pthread_mutex_lock
(3)	pthread_mutex_unlock
(4)	pthread_mutex_init
(5)	pthread_create
(6)	pthread_create
(7)	pthread_join
(8)	pthread_join

Grading guide: 1 mark for each blank.

   (b) (4 points) Explain what would happen if the lock is not used in the above code?

Which property of concurrent programs does the mutex ensure?

If the lock is not used, both threads modify **counter** simultaneously, leading to race condition.

The mutex ensures **mutual exclusion**, only one thread at a time updates counter.

Grading guide: 2 marks for each, 4 total.

(c) (3 points) Apart from concurrency, explain any three uses of Interprocess Communication (IPC).

Data sharing: In many applications, processes need to share data but each process has its own separate memory space. IPC provides mechanisms for processes to share data safely and efficiently.

Modular program design: Large programs can be divided into smaller, independent processes each handling a specific task. These processes can coordinate with each other using IPC mechanisms.

Resource sharing: Multiple processes may need access to the same resources (for eg., files). IPC provides mechanism to manage access to shared resources.

Distributed systems: Processes running on different machines need to communicate and exchange data. IPC mechanisms enable such communication.

Grading guide: 3 marks for any three of the above with explanation.

2. (15 points) You are designing the software for a smart coffee machine in an office. There are two types of threads:
Refiller thread: *refills coffee when the machine is empty.*
Drinker threads: *employees that want to drink coffee.*

The system must ensure the following: (1) A drinker can only drink if coffee is available. (2) If no coffee is available, the drinker waits until the refiller adds more. (3) The refiller only refills when the machine is empty (to avoid waste).

Use the following shared variables:

```
int coffee = 0;              // number of cups currently available
pthread_mutex_t lock;
pthread_cond_t cv_empty;     // signaled when machine becomes empty
pthread_cond_t cv_full;      // signaled when coffee is refilled
```

Assume that the **refiller** thread adds 5 cups at a time and that each **drinker** thread consumes 1 cup per drinking attempt. Complete the following C-code to synchronize these threads correctly using locks and condition variables.

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

int coffee = 0;
pthread_mutex_t lock = ---------------
pthread_cond_t cv_empty = ---------------
pthread_cond_t cv_full = ---------------

void *refiller(void *arg) {
    while (1) {
        ---------------
```

```
        ---------------           // wait until machine is empty
        ---------------

        coffee = 5;                   // refill
        printf("Refiller: refilled (5 cups)\n");
        pthread_cond_broadcast(&cv_full); // wake all waiting drinkers
        pthread_mutex_unlock(&lock);
    }
}

void *drinker(void *arg) {
    while (1) {
        ---------------
        ---------------           // wait until coffee available
        ---------------

        coffee--;
        printf("%s: drinking coffee (%d left)\n", (char *)arg, coffee);

        if (coffee == 0)
            pthread_cond_signal(&cv_empty); // wake refiller when empty

        pthread_mutex_unlock(&lock);
    }
}
```

Grading guide:

6 points for correct refiller and drinker threads each.

3 marks for correct initialization of the mutex lock and condition variables.

Total = 6+6+3 = 15.

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

int coffee = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t cv_full = PTHREAD_COND_INITIALIZER;

void *refiller(void *arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (coffee > 0)            // wait until machine is empty
            pthread_cond_wait(&cv_empty, &lock);

        coffee = 5;                   // refill
        printf("Refiller: refilled (5 cups)\n");
        pthread_cond_broadcast(&cv_full); // wake all waiting drinkers
        pthread_mutex_unlock(&lock);
    }
```

```
}

void *drinker(void *arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (coffee == 0)          // wait until coffee available
            pthread_cond_wait(&cv_full, &lock);

        coffee--;
        printf("%s: drinking coffee (%d left)\n", (char *)arg, coffee);

        if (coffee == 0)
            pthread_cond_signal(&cv_empty); // wake refiller when empty

        pthread_mutex_unlock(&lock);
    }
}
```

3. (10 points) Consider a scenario where a bus with capacity K, picks up waiting passengers from a bus stop. The bus arrives at the stop, allows up to K waiting passengers to board, and then departs. Passengers have to wait for the bus to arrive and then board it. Passengers who arrive at the bus stop after the bus has arrived should not be allowed to board and should wait for the next time the bus arrives. The bus and passengers are represented by threads in a program. The passenger thread should call the function **board()** after the passenger has boarded, and the bus should invoke **depart()** when it is ready to depart.

   The threads share the following variables, none of which are implicitly updated by functions like board() or depart().

```
mutex = semaphore initialized to 1.
bus_arrived = semaphore initialized to 0.
passenger_boarded = semaphore initialized to 0.
waiting_count = integer initialized to 0.
```

   Below is given synchronized code for the **passenger** thread.

```
sem_wait(mutex)
waiting_count++
sem_post(mutex)
sem_wait(bus_arrived)
board()
sem_post(passenger_boarded)
```

   (a) (8 points) Write and explain the corresponding synchronized code for the **bus thread**. The bus should board the correct number of passengers, based on its capacity and number of waiting by calling **sem_post/sem_wait** on semaphores. The bus code should also update the waiting count as required. Once boarding is completed, the bus thread should call **depart()**. You can use any extra local variables in the code of the bus thread, like integers and loop indices, but not

use any extra synchronization primitives.

```
sem_wait(mutex)
N = min(waiting_count, K)
for i = 1 to N
    sem_post(bus_arrived)
    sem_wait(passenger_boarded)
waiting_count = waiting_count - N
sem_post(mutex)
depart()
```

Grading guide:
1. 4 points for correct use of sem_wait and sem_post with correct order.
2. 1 point for correct for loop.
3. 2 points for correctly updating N and waiting_count.
4. 1 point for correctly calling depart().

(b) (2 points) Explain two types of non-deadlock bugs.

Atomicity-Violation Bugs: atomicity assumptions made by programmer are violated during execution of concurrent threads.
Order-Violation Bugs: desired order of accessing shared data is flipped during concurrent execution
(***Grading guide:*** 1 mark for each, with explanation.)