A friend of mine left her position teaching physics at FC College and took a job at PGCs. She was working in a cubicle in the basement of a huge building, and the nearest women's prayer room was two floors up. She proposed to her boss that women be allowed to pray in the men's pray room. The boss agreed, provided that the following synchronization constraints can be [illegible]ned: • There cannot be men and women in the prayer room at the same time. • There should never be [illegible]han three employees in the prayer room. You are supposed to write a synchronization solution for this problem and implement it in C language. You need to implement functions men() and women().

Code:

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

#define NUM_MEN 3
#define NUM_WOMEN 3

sem_t men_in_prayer_room;
sem_t women_in_prayer_room;
sem_t men_waiting;
sem_t women_waiting;

void* men(void* arg)
{
  while (1)
{
    sem_wait(&men_waiting);
    sem_wait(&men_in_prayer_room);
    // A man can now enter the prayer room
    printf("Man entering prayer room\n");
    sleep(1);
    // After a man is done praying, he leaves the prayer room
```

```
        printf("Man leaving prayer room\n");
        sem_post(&men_in_prayer_room);
        sem_post(&women_waiting);
    }
}
void* women(void* arg)
{
    while (1)
    {
        sem_wait(&women_waiting);
        sem_wait(&women_in_prayer_room);
        // A woman can now enter the prayer room
        printf("Woman entering prayer room\n");
        sleep(1);
        // After a woman is done praying, she leaves the prayer room
        printf("Woman leaving prayer room\n");
        sem_post(&women_in_prayer_room);
        sem_post(&men_waiting);
    }
}
int main()
{
    pthread_t men_threads[NUM_MEN];
    pthread_t women_threads[NUM_WOMEN];

    // Initialize semaphores
    sem_init(&men_in_prayer_room, 0, 0);
    sem_init(&women_in_prayer_room, 0, 0);
    sem_init(&men_waiting, 0, 0);
```

```
    sem_init(&women_waiting, 0, 0);
    // Create men threads
    for (int i = 0; i < NUM_MEN; i++)
    {
        pthread_create(&men_threads[i], NULL, men, NULL);
    }
    // Create women threads
    for (int i = 0; i < NUM_WOMEN; i++)
    {
        pthread_create(&women_threads[i], NULL, women, NULL);
    }
    // Wait for all threads to finish
    for (int i = 0; i < NUM_MEN; i++)
    {
        pthread_join(men_threads[i], NULL);
    }
    for (int i = 0; i < NUM_WOMEN; i++)
    {
        pthread_join(women_threads[i], NULL);
    }
    // Destroy semaphores
    sem_destroy(&men_in_prayer_room);
    sem_destroy(&women_in_prayer_room);
    sem_destroy(&men_waiting);
    sem_destroy(&women_waiting);
    return 0;
}
```

To vaccinate their students and respectable teachers, UCP is arranging a Covid-19 vaccination camp at the UCP. Management allowed one vaccination area for both teachers and students to

get their Covid-19 shot.

Management agreed, provided that the following synchronization constraints can be maintained:

- There cannot be students and teachers in the vaccination area at the same time.
- There should never be more than three people in the vaccination area.

You are supposed to write a synchronization solution for this problem and implement it in C Language

Code:

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

#define NUM_STUDENTS 10
#define NUM_TEACHERS 5
#define VACCINATION_CAPACITY 3

sem_t vaccination_room;
sem_t student_count;
sem_t teacher_count;

void *student_vaccination(void *arg)
{
  sem_wait(&student_count);
  sem_wait(&vaccination_room);

  // Student gets vaccinated here
  sem_post(&vaccination_room);
  sem_post(&student_count);
  return NULL;
}
void *teacher_vaccination (void *arg)
```

```
{
    sem_wait(&teacher_count);
    sem_wait(&vaccination_room);

    // Teacher gets vaccinated here
    sem_post(&vaccination_room);
    sem_post(&teacher_count);
    return NULL;
}
int main()
{
    sem_init(&vaccination_room, 0, VACCINATION_CAPACITY);
    sem_init(&student_count, 0, 0);
    sem_init(&teacher_count, 0, 0);

    pthread_t student_threads[NUM_STUDENTS];
    pthread_t teacher_threads[NUM_TEACHERS];

    for (int i = 0; i < NUM_STUDENTS; i++)
    {
        pthread_create(&student_threads[i], NULL, student_vaccination, NULL);
    }
    for (int i = 0; i < NUM_TEACHERS; i++)
    {
        pthread_create(&teacher_threads[i], NULL, teacher_vaccination, NULL);
    }
    for (int i = 0; i < NUM_STUDENTS; i++)
    {
        pthread_join(student_threads[i], NULL);
```

```
    }

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

    sem_destroy(&vaccination_room);
    sem_destroy(&student_count);
    sem_destroy(&teacher_count);
    return 0;
}
```

Thread synchronization using semaphores: The "NaCl" problem: You've been hired by Mother Nature to help with the chemical reaction to form Salt, which she doesn't seem to be able to get right due to synchronization problems. The trick is to get one Na (Sodium) atom and one Cl (Chlorine) atom all together at the same time. The atoms are threads. Each Na atom thread executes a procedure NReady() when it is ready to react; and each Cl atom thread invokes a procedure CReady() when it is ready.

Your job is to write the code for NReady() and CReady(). The procedures must delay until there are at least one Na atom and one Cl atom present, and then one of the procedures must call the procedure makeSalt(). After the makeSalt() call, instance of NReady() and instance of CReady() should return.

Code:

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

#define N_ATOMS 2

sem_t atomSemaphore[N_ATOMS];
sem_t saltSemaphore;
```

```
void NReady (int id)
{
    sem_wait(&atomSemaphore[id]);
    printf("Atom %d is ready\n", id);

    // Check if both Na and Cl atoms are present
    if (id == 0 && sem_getvalue(&atomSemaphore[1], NULL) > 0)
    {
        sem_post(&saltSemaphore);
    }
    else if (id == 1 && sem_getvalue(&atomSemaphore[0], NULL) > 0)
    {
        sem_post(&saltSemaphore);
    }
}
void CReady(int id)
 {
    sem_wait(&saltSemaphore);
    printf("Salt is ready\n");

    // Consume the atoms
    sem_post(&atomSemaphore[0]);
    sem_post(&atomSemaphore[1]);
}
void* atomThread(void* arg)
{
    int id = *(int*)arg;
    sem_init(&atomSemaphore[id], 0, 0);
```

```
    // Wait for a random amount of time before being ready
    sleep(rand() % 5);
    sem_post(&atomSemaphore[id]);

    // Wait until both atoms are ready to make salt
    NReady(id);
    return NULL;
}
void* saltThread()
{
    sem_init(&saltSemaphore, 0, 0);
    // Wait until both atoms are ready to make salt
    CReady(0);
    return NULL;
}
int main()
{
    pthread_t atomThreads[N_ATOMS];
    pthread_t saltThreadId;
    // Initialize the semaphores
    for (int i = 0; i < N_ATOMS; i++) {
        sem_init(&atomSemaphore[i], 0, 0);
    }
    sem_init(&saltSemaphore, 0, 0);

    // Create the atom threads
    for (int i = 0; i < N_ATOMS; i++)
    {
```

```
    int* id = malloc(sizeof(int));
    *id = i;
    pthread_create(&atomThreads[i], NULL, atomThread, id);
  }
  // Create the salt thread
  pthread_create(&saltThreadId, NULL, saltThread, NULL);

  // Wait for the threads to finish
  for (int i = 0; i < N_ATOMS; i++)
  {
    pthread_join(atomThreads[i], NULL);
  }
  pthread_join(saltThreadId, NULL);
  return 0;
}
```

Write a program that receives two command-line arguments, N and M, such that N >= M. The program calculates the factorial of N using the M number of threads. Each thread should display its product. The main thread calculates the final product.

Code:

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

// Global variables
int N, M;
unsigned long long result = 1;

// Function to calculate factorial for each thread
void *calculateFactorial(void *arg)
```

```
{
  int threadNumber = *((int *)arg);
  unsigned long long partialResult = 1;
  for (int i = threadNumber; i <= N; i += M)
  {
    partialResult *= i;
  }
  printf("Thread %d: Partial result = %llu\n", threadNumber, partialResult);
  // Update the global result
  pthread_mutex_lock(&mutex);
  result *= partialResult;
  pthread_mutex_unlock(&mutex);
  pthread_exit(NULL);
}
int main (int argc, char *argv[])
 {
  if (argc != 3)
  {
    fprintf(stderr, "Usage: %s N M\n", argv[0]);
    return EXIT_FAILURE;
  }
  N = atoi(argv[1]);
  M = atoi(argv[2]);

  if (N < M || N <= 0 || M <= 0)
  {
    fprintf(stderr, "Invalid input. N should be greater than or equal to M, and both should
be positive.\n");
    return EXIT_FAILURE;
   }
```

```
    pthread_t threads[M];
    int threadNumbers[M];
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

    // Create threads
    for (int i = 0; i < M; i++)
    {
        threadNumbers[i] = i + 1;
        pthread_create(&threads[i], NULL, calculateFactorial, (void *)&threadNumbers[i]);
    }
    // Join threads
    for (int i = 0; i < M; i++)
    {
        pthread_join(threads[i], NULL);
    }
    // Display final result
    printf("Final Result: %llu\n", result);
    return EXIT_SUCCESS;
}
```

Write a program that creates M worker threads to sum two M x N matrices and displays the resultant matrix. The program is passed values of M & N through command-line arguments. The program then initializes the values of matrices using the rand( ) function.

Code:

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

#define MAX_M 100
```

```
#define MAX_N 100

// Define the matrices and their dimensions
int A[MAX_M][MAX_N];
int B[MAX_M][MAX_N];
int C[MAX_M][MAX_N];
int M, N;

// Define a structure to hold thread parameters
struct ThreadParams {
    int row_start;
    int row_end;
};
// Function to be executed by each worker thread
void* matrixSum(void* param)
{
    struct ThreadParams* params = (struct ThreadParams*)param;

    for (int i = params->row_start; i < params->row_end; ++i)
    {
        for (int j = 0; j < N; ++j)
        {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
    pthread_exit(NULL);
}
int main(int argc, char* argv[])
{
    // Check if M and N are provided as command-line arguments
```

```
if (argc != 3)
{
    fprintf(stderr, "Usage: %s <M> <N>\n", argv[0]);
    return EXIT_FAILURE;
}

// Parse command-line arguments
M = atoi(argv[1]);
N = atoi(argv[2]);

// Check if dimensions are within limits
if (M <= 0 || N <= 0 || M > MAX_M || N > MAX_N)
{
    Fprintf (stderr, "Invalid dimensions. M and N should be positive and less than or equal to %d x %d\n", MAX_M, MAX_N);
    return EXIT_FAILURE;
}

// Initialize matrices A and B with random values
for (int i = 0; i < M; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        A[i][j] = rand() % 10;
        B[i][j] = rand() % 10;
    }
}
// Display matrix A
printf("Matrix A:\n");
for (int i = 0; i < M; ++i)
```

```
    for (int j = 0; j < N; ++j)

        printf("%d ", A[i][j]);
    }
    printf("\n");
}
// Display matrix B
printf("\nMatrix B:\n");
for (int i = 0; i < M; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        printf("%d ", B[i][j]);
    }
    printf("\n");
}

// Create M worker threads
pthread_t threads[M];
struct ThreadParams params[M];
int rows_per_thread = M / 2;  // Assuming M is even for simplicity

for (int i = 0; i < M; ++i)
{
    params[i].row_start = i * rows_per_thread;
    params[i].row_end = (i + 1) * rows_per_thread;
    pthread_create(&threads[i], NULL, matrixSum, (void*)&params[i]);
}
```

```
// Wait for all threads to finish
  for (int i = 0; i < M; ++i)
  {
     pthread_join(threads[i], NULL);
  }
  // Display the resultant matrix C
  printf("\nResultant Matrix C:\n");
  for (int i = 0; i < M; ++i)
  {
     for (int j = 0; j < N; ++j)
     {
        printf("%d ", C[i][j]);
     }
     printf("\n");
  }
  return EXIT_SUCCESS;
}
```

Write a C program where the parent process sends several numbers to its two child processes using shared memory. The child processes receive these numbers, compute their sum, and then send the result back to the parent process through the same shared memory. Subsequently, the parent process displays the final sum.

For reliable communication, you must implement the bounded buffer (i.e., given in the textbook) on shared memory. You need to use the semaphore library for synchronization.

The numbers that are to be passed to the child processes must be passed to the parent process as command-line arguments.

Assignment Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
```

```
#define BUFFER_SIZE 5

typedef struct
{
    int numbers[BUFFER_SIZE];
    int sum;
    sem_t empty;
    sem_t full;
    sem_t mutex;
}
SharedBuffer;

void initialBuffer(SharedBuffer *buffer)
{
    sem_init(&buffer->empty, 1, BUFFER_SIZE);
    sem_init(&buffer->full, 1, 0);
    sem_init(&buffer->mutex, 1, 1);
    buffer->sum = 0;
}

void producer(SharedBuffer *buffer, int *numbers, int count)
{
    for (int i = 0; i < count; ++i)
    {
        sem_wait(&buffer->empty);
        sem_wait(&buffer->mutex);

        buffer->numbers[i % BUFFER_SIZE] = numbers[i];

        sem_post(&buffer->mutex);
        sem_post(&buffer->full);
    }
}

void consumer(SharedBuffer *buffer)
{
    while (1)
    {
        sem_wait(&buffer->full);
        sem_wait(&buffer->mutex);

        int number = buffer->numbers[buffer->sum % BUFFER_SIZE];
        buffer->sum += number;

        sem_post(&buffer->mutex);
        sem_post(&buffer->empty);
    }
}

void cleanBuffer(SharedBuffer *buffer, int shmid)
```

```
{
    shmdt(buffer);
    shmctl(shmid, IPC_RMID, NULL);
    sem_destroy(&buffer->empty);
    sem_destroy(&buffer->full);
    sem_destroy(&buffer->mutex);
}

int main (int argc, char *argv[])
{
    int count = argc - 1;
    int *numbers = (int *)malloc(count * sizeof(int));

    for (int i = 0; i < count; ++i)
    {
        numbers[i] = atoi(argv[i + 1]);
    }

    key_t key = ftok(".", 'S');
    int shmid = shmget(key, sizeof(SharedBuffer), IPC_CREAT | 0666);
    SharedBuffer *buffer = (SharedBuffer *)shmat(shmid, NULL, 0);

    initialBuffer(buffer);
    pid_t pid1 = fork();

    if (pid1 == 0)
    {
        // Child 1
        consumer(buffer);
        exit(EXIT_SUCCESS);
    }
    pid_t pid2 = fork();
    if (pid2 == 0)
    {
        // Child 2
        consumer(buffer);
        exit(EXIT_SUCCESS);
    }

    // Parent process
    producer(buffer, numbers, count);

    // Ensure that producer has sent all numbers before terminating
    while (1)
    {
        int value = 0;
        sem_getvalue(&buffer->full, &value);
        if (value == 0)
        {
            break;
```

```
    }
  }

  // Print the sum of the input values
  printf("Sum of input values: %d\n", buffer->sum);

  // Cleanup
  cleanBuffer(buffer, shmid);
  free(numbers);

  return EXIT_SUCCESS;
}
```

Write a program to create two Threads T1 and T2. Thread T1 creates a file named Thread1.txt While T2 write "Hello T2 here" into theThread1.txt.

Code:

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

void* thread1_function(void* arg)
{
  // Thread T1: Create a file named Thread1.txt
  FILE *file = fopen("Thread1.txt", "w");

  if (file == NULL)
  {
    perror("Error creating the file");
    exit(EXIT_FAILURE);
  }

  fprintf(file, "File created by Thread T1\n");
  fclose(file);
  pthread_exit(NULL);
}
```

```
void* thread2_function(void* arg)
{
    // Thread T2: Write "Hello T2 here" into the Thread1.txt file
    FILE *file = fopen("Thread1.txt", "a");          // Open the file in append mode

    if (file == NULL)
    {
        perror("Error opening the file");
        exit(EXIT_FAILURE);
    }

    fprintf(file, "Hello T2 here\n");
    fclose(file);
    pthread_exit(NULL);
}
int main()
{
    pthread_t thread1, thread2;
    // Create Thread T1
    if (pthread_create(&thread1, NULL, thread1_function, NULL) != 0)
    {
        perror("Error creating Thread T1");
        exit(EXIT_FAILURE);
    }
    // Create Thread T2
    if (pthread_create(&thread2, NULL, thread2_function, NULL) != 0)
    {
        perror("Error creating Thread T2");
        exit(EXIT_FAILURE);
```

```
    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Threads T1 and T2 have finished their tasks.\n");
    return EXIT_SUCCESS;
}
```

Write a program to create a thread TI. The main process to passes two numbers to TI. TI calculates the sum of these numbers and returns the sum of to the parent process for printing.

Code:

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

// Structure to hold the parameters passed to the thread
struct ThreadParams
{
    int num1;
    int num2;
};
// Function to be executed by the thread T1
void* threadFunction(void* param)
{
    struct ThreadParams* params = (struct ThreadParams*)param;
    // Calculate the sum of the two numbers
    int sum = params->num1 + params->num2;

    // Return the result
    int* result = (int*)malloc(sizeof(int));
    *result = sum;
```

```
    pthread_exit(result);
}
int main()
{
    pthread_t thread1;
    struct ThreadParams params;
    // Get two numbers from the user
    printf("Enter the first number: ");
    scanf("%d", &params.num1);

    printf("Enter the second number: ");
    scanf("%d", &params.num2);
    // Create Thread T1 and pass the parameters
    if (pthread_create(&thread1, NULL, threadFunction, (void*)&params) != 0)
    {
        perror("Error creating Thread T1");
        exit(EXIT_FAILURE);
    }

    // Wait for Thread T1 to finish and get the result
    int* result;
    pthread_join(thread1, (void**)&result);
    // Print the result
    printf("Sum of %d and %d is: %d\n", params.num1, params.num2, *result);
    // Free the memory allocated for the result
    free(result);
    return EXIT_SUCCESS;
}
```

Write a program to create a thread T1. The main process to passestwo numbers to T1. T1 calculates the average of these numbers and returns the sum of to the parent process for printing.

Code:

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

// Structure to hold the parameters passed to the thread
struct ThreadParams
{
  int num1;
  int num2;
  double average;
};
// Function to be executed by the thread T1
void* threadFunction(void* param)
{
  struct ThreadParams* params = (struct ThreadParams*)param;

  // Calculate the average of the two numbers
  params->average = (params->num1 + params->num2) / 2.0;
  pthread_exit(NULL);
}
int main()
{
  pthread_t thread1;
  struct ThreadParams params;
  // Get two numbers from the user
  printf("Enter the first number: ");
  scanf("%d", &params.num1);
```

```
 printf("Enter the second number: ");
 scanf("%d", &params.num2);
 // Create Thread T1 and pass the parameters
 if (pthread_create(&thread1, NULL, threadFunction, (void*)&params) != 0)
{
    perror("Error creating Thread T1");
    exit(EXIT_FAILURE);
 }

 // Wait for Thread T1 to finish
 pthread_join(thread1, NULL);
 // Print the result
 printf("Average of %d and %d is: %.2f\n", params.num1, params.num2, params.average);

 return EXIT_SUCCESS;
}
```

Program-to-calculate-average-maximum-minimum

Code:

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

#define MAX_NUMBERS 50
struct ThreadData
 {
   int* numbers;
   int size;
  double average;
   int maximum;
```

```
  int minimum;
};
// function to calculate average of numbers
void* calculateAverage(void* data)
{
   struct ThreadData* threadData = (struct ThreadData*)data;
   for (int i = 0; i < threadData->size; ++i)
    {
      threadData->average += threadData->numbers[i];
    }
   threadData->average /= threadData->size;

   printf("Thread for average calculation : %lu \n", pthread_self());
   pthread_exit(NULL);
}
// function to find the maximum
void* calculateMaximum(void* data)
{
   struct ThreadData* threadData = (struct ThreadData*)data;
   threadData->maximum = threadData->numbers[0];

   for (int i = 1; i < threadData->size; ++i)
   {
     if (threadData->numbers[i] > threadData->maximum)
     {
       threadData->maximum = threadData->numbers[i];
     }
   }
   printf("Thread for maximum calculation : %lu \n", pthread_self());
```

```
    pthread_exit(NULL);
  }
  // function to find the minimum
  void* calculateMinimum(void* data)
  {
    struct ThreadData* threadData = (struct ThreadData*)data;
    threadData->minimum = threadData->numbers[0];

    for (int i = 1; i < threadData->size; ++i)
    {
      if (threadData->numbers[i] < threadData->minimum)
      {
        threadData->minimum = threadData->numbers[i];
      }
    }
    printf("Thread for minimum calculation : %lu \n", pthread_self());
    pthread_exit(NULL);
}
int main(int argc, char* argv[])
{
  if (argc < 2)
  {
    fprintf(stderr, "Usage: %s <num1> <num2> ... <numN>\n", argv[0]);
    return EXIT_FAILURE;
  }
 int size = argc - 1;
  if (size > MAX_NUMBERS)
 {
    fprintf(stderr, "Maximum numbers allowed are %d\n", MAX_NUMBERS);
```

```
    return EXIT_FAILURE;
}
// allocating memory for numbers array
int* numbers = (int*)malloc(size * sizeof(int));
if (numbers == NULL)
{
    perror("Got memory allocation error");
    return EXIT_FAILURE;
}
// parse command line arguments
for (int i = 0; i < size; ++i)
{
    numbers[i] = atoi(argv[i + 1]);
}
struct ThreadData data = {numbers, size, 0.0, 0, 0};
pthread_t threads[3];
// create threads
if (pthread_create(&threads[0], NULL, calculateAverage, (void*)&data) != 0 ||
    pthread_create(&threads[1], NULL, calculateMaximum, (void*)&data) != 0 ||
    pthread_create(&threads[2], NULL, calculateMinimum, (void*)&data) != 0)
{
    perror("Error creating threads");
    free(numbers);
    return EXIT_FAILURE;
}

// wait for threads to finish
for (int i = 0; i < 3; ++i)
{
```

```
        if (pthread_join(threads[i], NULL) != 0)
        {
            perror("Error joining threads");
            free(numbers);
            return EXIT_FAILURE;
        }
    }
    // print results
    printf("Average of Numbers: %.2f\n", data.average);
    printf("Maximum of Numbers: %d\n", data.maximum);
    printf("Minimum of Numbers: %d\n", data.minimum);
    // free allocated memory
    free(numbers);
    return EXIT_SUCCESS;
}
```

Multiplication of matrix

Code:

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

#define SIZE 3
int matrixA[SIZE][SIZE] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int resultMatrix[SIZE][SIZE];
struct ThreadData
{
    int row;
};
// Function to perform matrix multiplication for one row
```

```
void *multiplyRow(void *arg)
{
    struct ThreadData *data = (struct ThreadData *)arg;
    int row = data->row;

    for (int col = 0; col < SIZE; col++)
    {
        resultMatrix[row][col] = 0;
        for (int k = 0; k < SIZE; k++)
        {
            resultMatrix[row][col] += matrixA[row][k] * matrixA[k][col];
        }
    }
    pthread_exit(NULL);
}
int main()
{
    pthread_t threads[SIZE];
    struct ThreadData threadData[SIZE];
    // Create threads
    for (int i = 0; i < SIZE; i++)
    {
        threadData[i].row = i;
        pthread_create(&threads[i], NULL, multiplyRow, (void *)&threadData[i]);
    }
    // Join threads
    for (int i = 0; i < SIZE; i++)
    {
        pthread_join(threads[i], NULL);
```

```
    }
    // Print the result matrix
    printf("Result Matrix:\n");
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            printf("%d ", resultMatrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Implement a program that simulates a simple producer-consumer problem using hardware solutions for synchronization. The program should consist of one producer thread and one consumer thread. The producer thread writes to a shared buffer, and the consumer thread reads from the buffer and displays the data. The requirements for your program are as follows:

- **The producer thread should write to the buffer only if there is available space in the buffer.**
- **The consumer thread should read data from the buffer only if new data has been written by the producer thread. It must read the data in the same order it was written.**
- **If the buffer is full, the producer thread should wait until space becomes available.**
- **If the buffer is empty, the consumer thread should wait until new data is written by the producer thread.**
- **Use hardware solutions for synchronization, such as mutex or semaphores, to ensure prop coordination between the producer and consumer threads.**
- **Display appropriate messages to indicate when the producer writes to the buffer and when the consumer reads from the buffer.**

Code:

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

#define BUFFER_SIZE 5
int buffer[BUFFER_SIZE];
int count = 0;
pthread_mutex_t mutex;
sem_t full, empty;
void *producer(void *arg)
{
  for (int i = 1; i <= 10; i++)
  {
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
```

```
        buffer[count++] = i;
        printf("Producer wrote %d to the buffer.\n", i);
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}
void *consumer(void *arg)
{
    for (int i = 1; i <= 10; i++)
    {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        int item = buffer[--count];

        printf("Consumer read %d from the buffer.\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}
int main()
{
    pthread_t producerThread, consumerThread;
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
```

```
    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);
    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);
    return 0;
}
```

Write a program that takes three command-line arguments: a filename, a number N, and a string S. The program should open the specified file and search for the string S using N number of threads. Each thread should search a portion of the file. If any thread finds the string S, it should print a message indicating the line and column number and exit. The other threads should exit immediately once the string is found by any thread. Your program should handle errors appropriately, such as if the file cannot be opened, or if the specified number of threads is invalid.

Code:

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

#define MAX_STRING_SIZE 1000
struct ThreadArgs
{
    FILE *file;
    char *searchString;
    int threadID;
```

```
    int numThreads;
};
void *searchInFile(void *args)
{
    struct ThreadArgs *threadArgs = (struct ThreadArgs *)args;
    fseek(threadArgs->file, 0, SEEK_END);

    long fileSize = ftell(threadArgs->file);
    long chunkSize = fileSize / threadArgs->numThreads;

    fseek(threadArgs->file, threadArgs->threadID * chunkSize, SEEK_SET);
    char *buffer = (char *)malloc(MAX_STRING_SIZE);
    fread(buffer, 1, chunkSize, threadArgs->file);
    char *position = strstr(buffer, threadArgs->searchString);
    if (position != NULL)
    {
        int lineNumber = 1;
        for (char *c = buffer; c != position; c++)
        {
            if (*c == '\n')
            {
                lineNumber++;
            }
        }
        int columnNumber = position - buffer + 1;
        printf("String found at Line %d, Column %d by Thread %d\n", lineNumber, columnNumber,
threadArgs->threadID + 1);
```

```
            free(buffer);
            fclose(threadArgs->file);
            pthread_exit(NULL);
        }
        free(buffer);
        pthread_exit(NULL);
}
int main (int argc, char *argv[])
{
    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s <filename> <number_of_threads> <search_string>\n",.argv[0]);
        return EXIT_FAILURE;
    }
    char *filename = argv[1];
    int numThreads = atoi(argv[2]);
    char *searchString = argv[3];
    if (numThreads <= 0)
    {
        fprintf(stderr, "Invalid number of threads\n");
        return EXIT_FAILURE;
    }
    FILE *file = fopen(filename, "r");
    if (file == NULL)
     {
        perror("Error opening file");
        return EXIT_FAILURE;
```

```
    }
    pthread_t threads[numThreads];
    struct ThreadArgs threadArgs[numThreads];
    for (int i = 0; i < numThreads; i++)
    {
        threadArgs[i].file = file;
        threadArgs[i].searchString = searchString;
        threadArgs[i].threadID = i;
        threadArgs[i].numThreads = numThreads;

        pthread_create(&threads[i], NULL, searchInFile, &threadArgs[i]);
    }
    for (int i = 0; i < numThreads; i++)
    {
        pthread_join(threads[i], NULL);
    }
    fclose(file);
    return EXIT_SUCCESS;
}
```

Implement a program that simulates a simple producer-consumer problem using hardware solutions for synchronization. The program should consist of one producer thread and one consumer thread. The producer thread writes to a shared buffer, and the consumer thread reads from the buffer and displays the data. The requirements for your program are as follows:

- **The producer thread should write to the buffer only if there is available space in the buffer.**
- **The consumer thread should read data from the buffer only if new data has been written by the producer thread. It must read the data in the same order it was written.**
- **If the buffer is full, the producer thread should wait until space becomes available.**
- **If the buffer is empty, the consumer thread should wait until new data is written by the producer thread.**
- **Use hardware solutions for synchronization, such as mutex or semaphores, to ensure proper coordination between the producer and consumer threads.**
- **Display appropriate messages to indicate when the producer writes to the buffer and when the consumer reads from the buffer.**

Code:

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

#define BUFFER_SIZE 5

// Shared data between producer and consumer
int buffer[BUFFER_SIZE];
int count = 0; // Number of items in the buffer
// Mutex for synchronization
pthread_mutex_t mutex;
// Semaphores to signal buffer full/empty conditions
sem_t full, empty;
void *producer(void *arg)
```

```
{
  for (int i = 1; i <= 10; i++)
  {
    // Wait if the buffer is full
    sem_wait(&empty);
    pthread_mutex_lock(&mutex);
    // Produce item and add to the buffer
    buffer[count++] = i;
    printf("Producer wrote %d to the buffer.\n", i);
    pthread_mutex_unlock(&mutex);
    sem_post(&full);
  }
  pthread_exit(NULL);
}
void *consumer(void *arg)
{
  for (int i = 1; i <= 10; i++)
  {
    // Wait if the buffer is empty
    sem_wait(&full);
    pthread_mutex_lock(&mutex);
    // Consume item from the buffer
    int item = buffer[--count];
    printf("Consumer read %d from the buffer.\n", item);
    pthread_mutex_unlock(&mutex);
    sem_post(&empty);
  }
```

```
    pthread_exit(NULL);
}

int main()
{
    pthread_t producerThread, consumerThread;
    // Initialize mutex and semaphores
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);
    // Create producer and consumer threads
    pthread_create(&producerThread, NULL, producer, NULL);
    pthread_create(&consumerThread, NULL, consumer, NULL);
    // Join threads
    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);
    // Clean up resources
    pthread_mutex_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);

    return 0;
}
```