Assignment 7

Question

Consider a main process which creates three threads Th1, Th2, and Th3. The main process also creates two random quantities (X, Y), both less than 10. These two values will be placed by the main process in the shared memory (One variant of IPC Primitive) that is accessible by all the three threads Th1, Th2 and Th3. The shared memory will be created by the main process also. For each pair of values (X,Y), it is required that some computations will be done by various threads. The thread Th1 will compute A (X*Y) and the thread Th2 will compute B (X*Y)/2). Similarly, Th3 computes C (X+Y), Th2 again computes D ((X*Y)/(X+Y)), and finally Th1 computes E ((X+Y)(X-Y)). All these values are kept in the shared memory in a tabular fashion as shown below. The number of (X,Y) pairs will be taken as an argument from the CLI. It is the responsibility of the main process to populate required numbers of (X,Y)s in the shared memory. The program will only exit when all A,B,C etc. are computed for all given (X,Y) values. Before exiting, all (X,Y)s, As, Bs etc. should be displayed. INPUT - N, number of random pairs OUTPUT FORMAT - Pairs(X,Y) | A | B | C | D | E ----------------------------- (1, 2) | 2 | 1 | 3 | .66 | -3 ------------------------------ (4, 1) | 4 | 2 | 5 | .8 | 15

Solution

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

#define NUM_THREADS 3

// defining the structure for shared memory
struct SharedMemory {
        int x;      // X value
        int y;      // Y value
        double A;   // result of X * Y
        double B;   // result of (X * Y) / 2
        double C;   // result of X + Y
        double D;   // result of (X * Y) / (X + Y)
        double E;   // result of (X + Y)(X - Y)
};

// shared memory data structure
struct SharedMemory *shm;

// decalring the total number of pairs
int numPairs;

// Barrier for synchronization
pthread_barrier_t barrier;
```

```
// thread 1 computes A (X * Y) and E ((X + Y)(X - Y))
void *thread1() {
        for(int i = 0; i < numPairs; i++){
                pthread_barrier_wait(&barrier);
                shm[i].A = shm[i].x * shm[i].y;
                shm[i].E = (shm[i].x + shm[i].y) * (shm[i].x - shm[i].y);
        }
}

// thread 2 computes B (X * Y) / 2 and D ((X * Y) / (X + Y))
void *thread2() {
        for(int i = 0; i < numPairs; i++){
                pthread_barrier_wait(&barrier);
                shm[i].B = (shm[i].x * shm[i].y) / 2.0;
                shm[i].D = (shm[i].x * shm[i].y) / (shm[i].x + shm[i].y);
        }
}

// thread 3 computes C (X + Y)
void *thread3() {
        for(int i = 0; i < numPairs; i++){
                pthread_barrier_wait(&barrier);
                shm[i].C = shm[i].x + shm[i].y;
        }
}


int main(int argc, char *argv[]) {
        // checking if the correct number of arguments is provided
        if (argc != 2) {
                printf("Usage: %s <number_of_pairs>\n", argv[0]);
                exit(1);
        }

        numPairs = atoi(argv[1]); // to get the number of pairs from the command line
argument

        if (numPairs == 0) {
                fprintf(stderr, "Error: Invalid argument :: numPairs must be
positive\n");
                exit(EXIT_FAILURE);
        }

        key_t key = ftok("shmfile", 65); // generating a unique key for the shared
memory

        // creating a shared memory segment
        int shmid = shmget(key, numPairs * sizeof(struct SharedMemory), IPC_CREAT |
0666);
        if(shmid == -1){
                perror("shmget");
```

```
                exit(1);
        }

        // to attach shared memory
        shm = (struct SharedMemory *)shmat(shmid, NULL, 0);
        if(shm == (struct SharedMemory *)(-1)){
                perror("shmat");
                exit(1);
        }

        pthread_barrier_init(&barrier, NULL, 4); // initializing barrier

        pthread_t threads[NUM_THREADS];

        // creating threads to compute values for X * Y, (X * Y) / 2, and X + Y
        pthread_create(&threads[0], NULL, thread1, NULL);
        pthread_create(&threads[1], NULL, thread2, NULL);
        pthread_create(&threads[2], NULL, thread3, NULL);

        for(int i = 0; i < numPairs; i++){
                // generating random X and Y values
                shm[i].x = rand() % 9 + 1;
                shm[i].y = rand() % 9 + 1;
                pthread_barrier_wait(&barrier);
        }

        // waiting for all threads to complete
        for(int j = 0; j < NUM_THREADS; j++){
                pthread_join(threads[j], NULL);
        }

        pthread_barrier_destroy(&barrier); // destroying the barrier

        // displaying the computed results
        printf("Pairs(X,Y)\t|A\t|B\t|C\t|D\t|E\n");
        printf("---------------------------------------------------------\n");
        for(int i = 0; i < numPairs; i++){
                printf("Pairs(%d, %d) \t|%.2lf\t|%.2lf\t|%.2lf\t|%.2lf\t|%.2lf\n",
shm[i].x, shm[i].y, shm[i].A, shm[i].B, shm[i].C, shm[i].D, shm[i].E);
        }

        // creating command for ipcs system call
        char command[100];
        sprintf(command, "ipcs --id %d -m -l", shmid);

        // calling ipcs command to display shared memory stats
        int err = system(command);
        if(err != 0){
                perror("system");
                exit(1);
        }
```

```
    // detaching and removing shared memory
    shmdt(shm);
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

Explaination

1. **Header Inclusions**:

    - The program includes various header files that provide necessary functions and types. These headers include `<stdio.h>`, `<stdlib.h>`, `<unistd.h>`, `<pthread.h>`, `<sys/ipc.h>`, and `<sys/shm.h>`. These headers are needed for I/O, memory management, threading, and inter-process communication functionalities.

2. **Shared Memory Structure**:

    - The code defines a structure named `SharedMemory`. This structure contains several fields:
        - `x` and `y`: Integer values representing pairs of numbers.
        - `A`, `B`, `C`, `D`, and `E`: Double values to store the results of various computations.
    - This structure defines the format of data that will be stored in shared memory.

3. **Global Variables**:

    - `shm`: A pointer to the shared memory region. It will be used to access the shared data.
    - `numPairs`: An integer variable that represents the number of pairs for which mathematical computations will be performed.
    - `pthread_barrier_t barrier`: A synchronization barrier used to ensure that all threads start their computations simultaneously.

4. **Thread Functions**:

    - The code defines three thread functions, `thread1`, `thread2`, and `thread3`. Each of these functions is responsible for performing specific computations on the shared data.
    - `thread1` calculates `A` and `E` for each pair of `x` and `y`.
    - `thread2` calculates `B` and `D`.
    - `thread3` calculates `C`.
    - All thread functions use the same shared memory region `shm` and rely on the synchronization barrier.

5. **`main` Function**:

    - The `main` function serves as the program's entry point.
    - It begins by checking the number of command-line arguments (`argc`) to ensure that the program receives the expected number of arguments.

6. **Shared Memory Creation**:

- The code generates a unique key using `ftok` based on the filename "shmfile" and the project identifier 65. This key is used to identify the shared memory segment.
- It creates a shared memory segment using `shmget`. The size of the segment is determined by `numPairs * sizeof(SharedMemory)`. The `IPC_CREAT` flag indicates that a new segment should be created if it doesn't exist, and `0666` specifies the permissions.

7. **Shared Memory Attachment**:
   - The `shmat` function is used to attach the shared memory segment to the program's address space. The returned pointer `shm` now points to the shared memory region.
8. **Barrier Initialization**:
   - A synchronization barrier is initialized using `pthread_barrier_init`. The barrier ensures that all threads start their computations together.
9. **Thread Creation**:
   - Three threads (`thread1`, `thread2`, and `thread3`) are created using `pthread_create`. These threads will perform computations on the shared data.
10. **Random Value Generation**:
    - A loop generates random values for `x` and `y` for each pair. The values are generated using the `rand` function, and `numPairs` pairs are created.
11. **Barrier Synchronization**:
    - The program uses the synchronization barrier to ensure that all threads start their computations simultaneously. This prevents data races and ensures consistency.
12. **Thread Joining**:
    - After the threads have completed their computations, the program waits for all threads to finish using `pthread_join`.
13. **Barrier Destruction**:
    - The synchronization barrier is destroyed using `pthread_barrier_destroy` to release associated resources.
14. **Result Display**:
    - The computed results (`A`, `B`, `C`, `D`, and `E`) for each pair are displayed in a tabular format. This information helps visualize the outcomes of the mathematical computations.
15. **Shared Memory Stats Display**:
    - The program constructs a command to call the `ipcs` utility with the `--id` option to display shared memory statistics for the shared memory segment. This provides information about the shared memory's current state.
16. **Shared Memory Cleanup**:

- The program detaches from the shared memory segment using `shmdt`, ensuring that it is no longer accessible from the program. It then removes the shared memory segment using `shmctl` with `IPC_RMID` to release the shared memory resources completely.

17. **Exit**:
    - Finally, the program exits with a status of 0 to indicate successful execution.

1. Shared Memory Creation (Code Section):

`key_t key = ftok("shmfile", 65);`

In the provided code, this line is used to generate a unique key that will be associated with the shared memory segment. The key is essential for identifying and accessing the shared memory segment. Here's a breakdown of how it works:

1. `ftok` Function:
    - `ftok` stands for "File to Key." It is a function available in Unix-like operating systems (including Linux) and is commonly used to generate keys for inter-process communication mechanisms such as shared memory and message queues.
2. Arguments:
    - `ftok` takes two arguments:
        - The first argument is a filename (in this case, "shmfile").
        - The second argument is a project identifier (usually a non-negative integer). In the code, it's provided as `65`.
3. Generating the Key:
    - `ftok` combines the given filename and project identifier to generate a unique 32-bit integer key.
    - It does this by using a hashing algorithm that converts the filename and project identifier into a unique key. The algorithm ensures that different filename-project identifier combinations result in different keys.
4. Key Uniqueness:
    - The uniqueness of the key is crucial because it allows different processes to refer to the same shared memory segment using the same key.
5. Shared Memory Identification:
    - Once the key is generated, it can be used when creating, accessing, or attaching to shared memory segments.
    - Other processes that need to access the same shared memory segment can generate the same key using the same filename and project identifier. This ensures they can locate and attach to the correct shared memory segment.
6. Important Considerations:
    - The filename used with `ftok` should correspond to an existing file in the filesystem. If the file does not exist, `ftok` will return an error.

    - The project identifier (65 in this case) should be chosen carefully to avoid collisions with other keys used in the system.

In summary, `key_t key = ftok("shmfile", 65);` generates a unique key based on the provided filename ("shmfile") and project identifier (65). This key is crucial for identifying and accessing the shared memory segment, ensuring that multiple processes can refer to the same shared memory using the same key.

`int shmid = shmget(key, numPairs * sizeof(struct SharedMemory), IPC_CREAT | 0666);`

The line `int shmid = shmget(key, numPairs * sizeof(struct SharedMemory), IPC_CREAT | 0666);` in the provided code is responsible for creating a new shared memory segment (or retrieving an existing one if it already exists). Let's break down this line of code:

1. `int shmid`:
    - This declares an integer variable named `shmid` to store the unique identifier (ID) of the shared memory segment once it is created or accessed.
2. `shmget` Function:
    - `shmget` is a system call in Unix-like operating systems (including Linux) that is used to create and manage shared memory segments.
    - It takes three arguments:
        - `key`: The unique key associated with the shared memory segment, which was generated using `ftok`.
        - `size_t size`: The size of the shared memory segment in bytes. In this case, it's calculated as `numPairs * sizeof(struct SharedMemory)`. It's the total size required for storing `numPairs` instances of the `struct SharedMemory`.
        - `int shmflg`: Flags that control the behavior of `shmget`. In this case, `IPC_CREAT | 0666` is used:
            - `IPC_CREAT`: This flag indicates that the shared memory segment should be created if it doesn't already exist.
            - `0666`: This is the permission mode for the shared memory segment, specified as an octal value. In this case, it grants read and write permissions to the owner and the group.
3. Shared Memory Creation:
    - If a shared memory segment with the specified key does not exist, `shmget` creates a new one with the specified size and permissions.
    - If a shared memory segment with the specified key already exists, `shmget` retrieves its identifier.
4. Error Handling:
    - If there is an error during shared memory creation or retrieval, `shmget` returns -1, and an error message can be obtained using `perror`.
5. `shmid` Value:
    - If the shared memory segment is successfully created or retrieved, its unique identifier (ID) is stored in the `shmid` variable.

In summary, this line of code creates a shared memory segment (or retrieves an existing one) with a specified key, size, and permissions. The `shmid` variable is then used to identify and access this shared memory segment in the program.

2. Shared Memory Attachment (Code Section):

The line `shm = (struct SharedMemory *)shmat(shmid, NULL, 0);` in the provided code is responsible for attaching (or mapping) the shared memory segment identified by `shmid` to the program's address space. Let's break down this line of code:

1. `shm`:

    - `shm` is a pointer to a structure of type `struct SharedMemory`. This pointer will be used to access and manipulate the data stored in the shared memory segment.

2. `shmat` Function:

    - `shmat` is a system call in Unix-like operating systems (including Linux) that is used to attach (map) a shared memory segment to the address space of a process.
    - It takes three arguments:
        - `int shmid`: The unique identifier (ID) of the shared memory segment to be attached. This ID was obtained previously using `shmget`.
        - `void *shmaddr`: The address at which the shared memory segment should be attached. In this case, `NULL` is passed, indicating that the system should choose a suitable address automatically.
        - `int shmflg`: Flags that control the attachment behavior. In this case, `0` is used, which means no special flags are set.

3. Shared Memory Attachment:

    - When `shmat` is called, it attaches the shared memory segment identified by `shmid` to the address space of the calling process.
    - If `shmaddr` is `NULL`, the system selects a suitable address for attachment.
    - The return value of `shmat` is the starting address of the shared memory segment within the process's address space. In this case, it's cast to a pointer of type `struct SharedMemory *` and assigned to the `shm` pointer.

4. Error Handling:

    - If there is an error during the attachment process, `shmat` returns `(void *)-1`, indicating failure.

In summary, this line of code attaches the previously created (or retrieved) shared memory segment to the program's address space, making it accessible for reading and writing. The `shm` pointer is then used to access and manipulate the data within the shared memory segment as if it were a regular C data structure.

3. Barrier Initialization (Code Section):

The line `pthread_barrier_init(&barrier, NULL, 4);` is initializing a pthread barrier named `barrier`. Let me break down what this line of code does:

- `pthread_barrier_init`: This is a function provided by the pthreads library for initializing a barrier. A barrier is a synchronization primitive that allows multiple threads to wait until all of them have reached a certain point in their execution before they proceed further.

- `&barrier`: This is a pointer to the barrier variable that you want to initialize. The `barrier` variable should be of type `pthread_barrier_t`, which is a pthreads barrier object.

- `NULL`: This argument is typically used to specify attributes for the barrier, but in this case, it's set to `NULL` to use the default attributes.

- `4`: This argument specifies the number of threads that must call `pthread_barrier_wait` on the `barrier` before they can all proceed. In this case, you are specifying that four threads must reach the barrier before they can continue.

So, this line of code initializes a barrier named `barrier` and sets it up so that it will block until all four threads have called `pthread_barrier_wait` on it. This can be used to synchronize the threads in your program, ensuring that they all reach a certain point before moving on.

4. Barrier Synchronization (Code Section):

- After generating random values for `x` and `y` in the loop, the program calls `pthread_barrier_wait(&barrier);`. This statement is used to synchronize the threads.
- The barrier ensures that all threads reach this point before any of them proceed further. It guarantees that all threads start their computations on the same set of data.

5. Thread Joining (Code Section):

- After creating the threads, the program waits for each thread to complete its execution using `pthread_join`.
- `pthread_join(threads[j], NULL);` waits for the thread with the identifier `threads[j]` to finish its execution.
- This ensures that the program doesn't proceed beyond this point until all threads have completed their computations.

6. Barrier Destruction (Code Section):

- Once all threads have completed their computations and the barrier is no longer needed, the program destroys the synchronization barrier using `pthread_barrier_destroy`.
- `pthread_barrier_destroy(&barrier);` releases any associated resources and allows the program to exit cleanly.