CS347 Lab 3

Shubham Kar, 180070058

April 2021

Problem Statement

Design various synchronisation primitives for thread execution using pthread library. The synchronisation primitives should be used to control the execution of a group of threads as applicable by the logic used by the primitive. The correctness of the code should be maintained even after using multithreading in a program.

Custom Defined Variables

Each synchronisation primitive uses four custom defined structures as variable types. The t_arg variable type is used to pass the function arguments to the respective functions after wrapping it up to include the thread ID it is currently in and the actual arguments needed by the thread as shown below:

```
typedef struct __t_arg_t {
        int t_arg_1;
        void *t_arg_2;
        int t_arg_3;
} t_arg;
```

It is not meant to be used by the user and is used by functions defined in the header files.
The custom defined variable synch is used to store the synchronization specific variables in it like the mutex and the condition variable. It also is used to store variables that are used for synchronisation and can vary from one primitive to the other like in thread ordering, it holds an array specified by the user for the order of thread execution whereas in barrier synchronization, it holds a variable storing the number of threads and a counter which gets updated once a thread reaches the barrier. This variable is not meant to be used by the user and is only used by the functions in the header. A sample definition for priority synchronisation is shown in the below listing:

```
typedef struct  __sync{
        int counter;
        int *priority_vec;
        int *priority_orig;
        pthread_cond_t cond;
        pthread_mutex_t lock;
} synch;
```

The custom defined variable thread_sync is supposed to be used by the user to initialize the respective synchronisation primitive. This variable wraps a pointer to a pthread_t variable which stores first element of the pthread_t array that the user wants to synchronize. thread_sync also wraps a synch variable which is used for synchronizing the array of threads pointed to by thread. The definition is common for all primitives and is shown in the below listing:

```
typedef struct __mythread{
        pthread_t *thread;
        synch sync_t;
} thread_sync;
```

The custom defined variable my_arg is supposed to be used by the user to supply arguments and pointers to functions to the corresponding threads which they want to run. It also holds a pointer to the synch variable in use by the thread_sync variable associated with the threads. Its structure is shown in the below listing:

```
typedef struct __myarg_t {
        void *(*routine)(void *);
        synch* sync_t;
        void *args;
        int tid;
} my_arg;
```

Thread Ordering

In this synchronisation problem, we want each thread created by the user to have an ID specified by the user. The user is also responsible for giving the order in which the threads should be executed as per their ID. The user also has to specify the function each process is going to run and also needs to provide the arguments for the function called for each thread. These can be passed using an array of pointers to function and their arguments.
First of all, the user has to include the **thread_order** header file. Next, he/she to create a **thread_synch** variable to be passed to the function **th_create** for creating the array of threads for synchronization. Then, the user has to pass pointers to the function he/she wants to run on the threads numbered from **0 to (n_threads-1)** and references to the arguments he/she wants each of those functions to use respectively. For doing this, the user needs to first define an **array of my_arg variables**. Then, he/she has to pass a pass the pointers to functions and the references to arguments using the **routine** and **args** field of

my_arg variable.
Then, the user has to pass the **thread_sync** variable, the reference to the **my_arg array** and the number of threads **n_threads** that they want to synchronize. They also need to pass the order of thread execution as an array with the execution being from thread ID in element 0 to thread ID of the last element of the array. Thread IDs go from **0 to (n_thread - 1)**. A sample main file running 8 threads with an arbitrary order defined beforehand is shown in Listing 1.

```
#include "thread_order.h"

void *myfunc(void *args){
        t_arg *t_args = (t_arg*)(args);
        int *twarg = (int *)(t_args->t_arg_2);
        printf("Hello...you are in thread %d with argument %d\n", t_args->t_arg_1, *twarg);
        return NULL;
}

int main(){
        int n_threads = 8;
        thread_sync threads;
        my_arg my_args[n_threads];

        int arg_vec[n_threads];
        int order_vec[] = {7, 2, 1, 4, 3, 0, 6, 5};

        for(int i=0; i<n_threads; i++){
                arg_vec[i] = i*i;
                my_args[i].routine = myfunc;
                my_args[i].args = &arg_vec[i];
        }

        th_create(&threads, my_args, order_vec, n_threads);
        th_join(&threads, order_vec, n_threads);

        return 0;
}
```

Listing 1: Sample main.c file using**thread_order** primitive

In the **th_create** function, the **initialize_sync** function is called first to initialize the **synch** variable of the **thread_sync** variable passed. Next, **n_threads** threads are created and the threads are all given their own ID from **0 to (n_threads - 1)**.
One important thing to note is that we are not calling the functions passed by the user during the thread creation. We are instead running a wrapper function

first which makes the created thread wait until all the other threads are created and then starts execution once the start flag of the **sync_t** becomes true and the thread currently awake is to be ran according to the order of execution. To do this, we make use of a mutex lock and a condition variable. In the wrapper function, we first acquire the lock and check whether all threads have finished being created or not. If not, the thread is put to sleep after signaling the condition variable **cond**. If yes, then it checks whether the thread ID is equal to the corresponding thread ID specified by the user in the order of execution. If not, it again goes to sleep and if yes, then it proceeds to wrap the arguments of the function and the thread ID in a variable **t_arg**. This is then passed to the corresponding function for the thread.
Using **t_arg_1** field of the variable is not optional here. It has just been used here to show the correctness of the implementation of thread ordering. The execution is started once all the threads are created and then the start flag of **sync_t** is enabled in **exec_my_threads** function. The various functions implementing this thread ordering is shown in the below listing:

```
typedef struct  __sync{
        int counter;
        int *id_vec;
        int start;
        pthread_cond_t cond;
        pthread_mutex_t lock;
} synch;

void *wrapper_func(void *args){
        my_arg *my_args = (my_arg*)(args);
        synch *sync_t = my_args->sync_t;

        pthread_mutex_lock(&(sync_t->lock));
        while((*(sync_t->id_vec + sync_t->counter) != my_args->tid) || (sync_t->start == 0))
                // printf("%d %d %d\n",sync_t->id_vec[sync_t->counter], my_args -> tid, sync
                pthread_cond_signal(&(sync_t->cond));
                pthread_cond_wait(&(sync_t->cond), &(sync_t->lock));
        }

        void *myfunc_arg = my_args-> args;
        void *(*routine)(void *) = my_args->routine;
        t_arg myfunc_targ;
        myfunc_targ.t_arg_1 = my_args->tid;
        myfunc_targ.t_arg_2 = myfunc_arg;
        routine(&myfunc_targ);

        (sync_t->counter)++;
        pthread_cond_signal(&(sync_t->cond));
        pthread_mutex_unlock(&(sync_t->lock));
        return NULL;
}

void initialize_synch(synch* sync_t, int n_threads, int order_vec[]){
        sync_t->counter = 0;
        sync_t->start = 0;
        int lc = pthread_mutex_init(&(sync_t->lock), NULL);
        assert(lc == 0);
        lc = pthread_cond_init(&(sync_t->cond), NULL);
        sync_t->id_vec = order_vec;
}

void exec_my_threads(synch* sync_t){
        pthread_mutex_lock(&(sync_t->lock));
        sync_t->start = 1;
        pthread_cond_signal(&(sync_t->cond));
        printf("Started Execution: %d\n", *(sync_t->id_vec + sync_t->counter));
        pthread_mutex_unlock(&(sync_t->lock));
}

void th_create(thread_sync *threads, my_arg my_args[], int order_vec[], int n_threads){
        initialize_synch(&(threads->sync_t), n_threads, order_vec);
        threads->thread = (pthread_t *)malloc(n_threads*sizeof(pthread_t));
        for(int i=0; i<n_threads; i++){
                my_args[i].sync_t = &(threads->sync_t);
                my_args[i].tid = i;
                 pthread_create((threads->thread + i), NULL, wrapper_func, &my_args[i]);
        }
        exec_my_threads(&(threads->sync_t));
}

void th_join(thread_sync *threads, int order_vec[], int n_threads){
        pthread_join(*(threads->thread + order_vec[n_threads-1]), NULL);
        free(threads->thread);
}
```

The output after running the main function shown before is as shown below:

```
Started Execution: 7
Hello...you are in thread 7 with argument 49
Hello...you are in thread 2 with argument 4
Hello...you are in thread 1 with argument 1
Hello...you are in thread 4 with argument 16
Hello...you are in thread 3 with argument 9
Hello...you are in thread 0 with argument 0
Hello...you are in thread 6 with argument 36
Hello...you are in thread 5 with argument 25
```

Clearly, the order of execution is being followed and since dummy arguments of i^2 were provided, the arguments have been not shared among the threads as apparent from the output.

Barrier Synchronization

In barrier synchronization, a barrier function is meant to be called by the user in the functions meant to be ran by the threads created. Using this function makes the thread wait for the all of the other threads to reach that point of execution where the particular barrier function is used. It is not meant to be used with a group of threads in which some threads may not call the barrier function and the other threads call it.
A sample main.c file for running the barrier synchronisation primitive is shown in Listing 2.

In the **barrier_synch** function, the thread first tries to acquire the lock given in **sync_t**. Once acquired, it implies that the thread has reached the point in the process where the**barrier_synch** function has been called successfully. Therefore, the count of threads waiting at this point is increased by one. Then, it checks whether the count is equal to total number of threads or not. If no, then the thread signals the condition variable **cond** and goes to sleep, and if yes, then the threads resume their operation after the function normally. No wrapper function is required in this case as was required in thread ordering. No such condition is placed that the execution of threads starts only after all threads are created as was placed in thread ordering to avoid unnecessary deadlock scenarios.

```
#include "barrier.h"

void *myfunc(void *args){
        my_arg *m_args = (my_arg*)(args);
        int tid = m_args->tid;
        int *myfunc_args = (int *)(m_args->args);

        printf("Hello...you entered thread %d with argument %d\n", tid, *myfunc_args);
        printf("Now we enter a barrier\n");

        barrier_synch(m_args->sync_t);

        printf("Barrier crossed for thread %d\n", tid);

        return NULL;
}

int main(){
        int n_threads = 8;
        thread_sync threads;
        my_arg my_args[n_threads];

        int arg_vec[8];

        for(int i=0; i<n_threads; i++){
                arg_vec[i] = i*i;
                my_args[i].args = &arg_vec[i];
                my_args[i].routine = myfunc;
        }

        mythread_create(&threads, my_args, n_threads);
        mythread_join(&threads, n_threads);

        return 0;
}
```

Listing 2: Sample main.c file for using barrier synchronisation primitive

The functions used in the barrier synchronization primitive header files are shown in the below listing:

```
typedef struct  __sync{
        int count;
        int n_threads;
        pthread_cond_t cond;
        pthread_mutex_t lock;
} synch;

void barrier_synch(synch* sync_t){
        pthread_mutex_lock(&(sync_t->lock));
        (sync_t->count)++;
        while(sync_t->count != sync_t->n_threads){
                // printf("%d %d\n", sync_t->count, sync_t->n_threads);
                pthread_cond_signal(&(sync_t->cond));
                pthread_cond_wait(&(sync_t->cond), &(sync_t->lock));
        }
        pthread_cond_signal(&(sync_t->cond));
        pthread_mutex_unlock(&(sync_t->lock));
}


void initialize_synch(synch* sync_t, int n_threads){
        sync_t->count = 0;
        sync_t->n_threads = n_threads;
        int lc = pthread_mutex_init(&(sync_t->lock), NULL);
        assert(lc == 0);
        lc = pthread_cond_init(&(sync_t->cond), NULL);
        assert(lc==0);
}

void mythread_create(thread_sync *threads, my_arg my_args[], int n_threads){
        initialize_synch(&(threads->sync_t), n_threads);
        threads->thread = (pthread_t *)malloc(n_threads*sizeof(pthread_t));
        for(int i=0; i<n_threads; i++){
                my_args[i].sync_t = &(threads->sync_t);
                my_args[i].tid = i;
                 pthread_create((threads->thread + i), NULL,
                 my_args[i].routine, &my_args[i]);
        }
}

void mythread_join(thread_sync *threads, int n_threads){
        for(int i=0; i<n_threads; i++){
                 pthread_join(*(threads->thread + i), NULL);
        }
        free(threads->thread);
}
```

The output for the main function shown in Listing 2 is shown in the below listing:

```
Hello...you entered thread 0 with argument 0
Now we enter a barrier
```

```
Hello...you entered thread 3 with argument 9
Now we enter a barrier
Hello...you entered thread 1 with argument 1
Now we enter a barrier
Hello...you entered thread 2 with argument 4
Now we enter a barrier
Hello...you entered thread 6 with argument 36
Now we enter a barrier
Hello...you entered thread 7 with argument 49
Now we enter a barrier
Hello...you entered thread 4 with argument 16
Now we enter a barrier
Hello...you entered thread 5 with argument 25
Now we enter a barrier
Barrier crossed for thread 7
Barrier crossed for thread 4
Barrier crossed for thread 5
Barrier crossed for thread 2
Barrier crossed for thread 3
Barrier crossed for thread 0
Barrier crossed for thread 1
Barrier crossed for thread 6
```

We observe that all the threads sleep once they encounter a barrier and wait for all the threads to reach that particular barrier. Once all of them have encountered the barrier, they cross it.

Priority Synchronization

In priority synchronization, similar to the thread ordering, an array has to be provided by the user indicating the priority levels of threads that are supposed to be executed. Threads with the higher priority will execute before the threads with lower priority. The process to create the threads is similar to the one in thread ordering primitive. A sample main.c file using the priority synchronization primitive is shown in Listing 3.

In the **mythread_create** function, first the**initialize_synch** function is called. In this function, the priority vector is first sorted in an ascending order. Then **n_threads** number of threads are created and a wrapper function is called where the execution of threads are handled wherein the threads with the highest priority are executed first and then the threads with the lower priority are executed. If two or more threads have the same priority, then the order of their execution cannot be determined exactly. The functions used to implement the priority synchronization primitive are shown in the below listing:

```
typedef struct  __sync{
        int counter;
```

```c
#include "priority.h"

void *myfunc(void *args){
        t_arg *t_args = (t_arg*)(args);
        int *twarg = (int *)(t_args->t_arg_2);
        printf("Hello...you are in thread %d with argument %d with priority %d\n",
        t_args->t_arg_1, *twarg, t_args->t_arg_3);
        return NULL;
}

int main(){
        int n_threads = 8;
        thread_sync threads;
        my_arg my_args[n_threads];

        int arg_vec[n_threads];
        int priority_vec[] = {1, 1, 2, 2, 1, 3, 2, 3};

        for(int i=0; i<n_threads; i++){
                arg_vec[i] = i*i;
                my_args[i].routine = myfunc;
                my_args[i].args = &arg_vec[i];
        }

        mythread_create(&threads, my_args, priority_vec, n_threads);
        mythread_join(&threads, n_threads);

        return 0;
}
```

Listing 3: Sample main.c file using the priority synchronization primitive

```c
        int *priority_vec;
        int *priority_orig;
        pthread_cond_t cond;
        pthread_mutex_t lock;
} synch;

typedef struct __mythread{
        pthread_t *thread;
        synch sync_t;
} thread_sync;

typedef struct __myarg_t {
        void *(*routine)(void *);
        synch* sync_t;
        void *args;
        int tid;
} my_arg;

void *wrapper_func(void *args){
        my_arg *my_args = (my_arg*)(args);
        synch *sync_t = my_args->sync_t;
        int tid = my_args->tid;

        pthread_mutex_lock(&(sync_t->lock));
        while(*(sync_t->priority_orig + tid) !=
        *(sync_t->priority_vec + sync_t->counter)){
                pthread_cond_signal(&(sync_t->cond));
                pthread_cond_wait(&(sync_t->cond), &(sync_t->lock));
        }

        void *myfunc_arg = my_args-> args;
        void *(*routine)(void *) = my_args->routine;
        t_arg myfunc_targ;
        myfunc_targ.t_arg_1 = tid;
        myfunc_targ.t_arg_2 = myfunc_arg;
        myfunc_targ.t_arg_3 = sync_t->priority_orig[tid];
        routine(&myfunc_targ);

        (sync_t->counter)--;
        pthread_cond_signal(&(sync_t->cond));
        pthread_mutex_unlock(&(sync_t->lock));

        return NULL;
}

void initialize_synch(synch *sync_t, int n_threads, int priority_vec[]){
        sync_t->counter = 0;
        sync_t->counter = (n_threads - 1);
        int lc = pthread_mutex_init(&(sync_t->lock), NULL);
        assert(lc == 0);
        lc = pthread_cond_init(&(sync_t->cond), NULL);
        assert(lc == 0);
        sync_t->priority_vec = priority_vec;
        sync_t->priority_orig = (int *)malloc(n_threads*sizeof(int));
        for(int i=0; i<n_threads; i++){
                *(sync_t->priority_orig + i) = priority_vec[i];
        }
        int temp;
        for(int i=0; i<n_threads; i++){
                for(int j=i+1; j<n_threads; j++){
                        if(*(sync_t->priority_vec + j)<=*(sync_t->priority_vec + i)){
                                temp = *(sync_t->priority_vec + j);
                                *(sync_t->priority_vec + j) =
                                *(sync_t->priority_vec + i);
                                *(sync_t->priority_vec + i) = temp;
                        }
                }
        }
}

void mythread_create(thread_sync *threads,
my_arg my_args[],
int priority_vec[],
int n_threads){
        initialize_synch(&(threads->sync_t), n_threads, priority_vec);
        threads->thread = (pthread_t *)malloc(n_threads*sizeof(pthread_t));
        for(int i=0; i<n_threads; i++){
                my_args[i].sync_t = &(threads->sync_t);
                my_args[i].tid = i;
                 pthread_create((threads->thread + i), NULL, wrapper_func, &my_args[i]);
        }
}

void mythread_join(thread_sync *threads, int n_threads){
        for(int i=0; i<n_threads; i++){
                 pthread_join(*(threads->thread + i), NULL);
        }
        free(threads->thread);
        synch *sync_t = &threads->sync_t;
        free(sync_t->priority_orig);
}
```

The output of the sample main function shown in listing 3 is shown in the below listing:

```
Hello...you are in thread 7 with argument 49 with priority 3
Hello...you are in thread 5 with argument 25 with priority 3
Hello...you are in thread 6 with argument 36 with priority 2
Hello...you are in thread 3 with argument 9 with priority 2
Hello...you are in thread 2 with argument 4 with priority 2
Hello...you are in thread 1 with argument 1 with priority 1
Hello...you are in thread 4 with argument 16 with priority 1
Hello...you are in thread 0 with argument 0 with priority 1
```

We observe that the correct execution is followed wherein the threads with higher priority are getting executed before the threads with lower priority.

Selective Barrier Synchronization

One disadvantage of Barrier synchronisation was that the threads needed to stop at a barrier until and unless all the threads encountered the barrier. It is possible for a situation in which in a set of threads, there may be some threads(Set 1) whose execution after a certain point is completely dependent on a value calculated by some other set of threads(Set 2). However, those Set 2 of threads do not depend on Set 1 of threads execution. using a barrier synchronisation primitive will hinder the functioning of the Set 2 of threads.
Selective Barrier synchronization primitive solves this very problem in which using 2 functions(for identifying the 2 set of threads), one can make one set of threads wait until the other set of threads complete execution upto a certain point. Those other set of threads are however not stopped in their execution after they reach that certain point. The former is implemented using selective_barrier_wait function and the latter using selective_barrier_signal function. The user is supposed to provide the total number of threads, the number of threads in Set 2, pointer to functions to be executed by respective threads and their arguments. A sample main.c file implementing selective barrier synchronization is shown in Listing 4.

The selective_barrier_wait function checks whether all the number of threads that finished their execution upto the certain mark in Set 2 of threads. If no, the thread in set 1 is put to sleep after signaling another thread and if yes, then the barrier in Set 1 is crossed for all the threads. The selective_barrier_signal function increments the count that is compared to the total number of threads in Set 2 in the selective_barrier_wait function. The functions used in implementing the selective barrier synchronization primitive is shown in the below listing:

```
typedef struct  __sync{
        int flag;
        int count;
        int n_threads2;
        pthread_cond_t cond;
        pthread_mutex_t lock;
} synch;

void selective_barrier_wait(synch *sync_t){
        pthread_mutex_lock(&(sync_t->lock));
        while(sync_t->flag == 0){
                // printf("%d %d\n", sync_t->count, sync_t->n_threads);
                pthread_cond_signal(&(sync_t->cond));
                pthread_cond_wait(&(sync_t->cond), &(sync_t->lock));
        }
        pthread_cond_signal(&(sync_t->cond));
        pthread_mutex_unlock(&(sync_t->lock));
}
```

```
void selective_barrier_signal(synch *sync_t){
        pthread_mutex_lock(&(sync_t->lock));
        (sync_t->count)++;
        if(sync_t->count == sync_t->n_threads2){
                sync_t->flag = 1;
        }
        pthread_cond_signal(&(sync_t->cond));
        pthread_mutex_unlock(&(sync_t->lock));
}

void initialize_synch(synch* sync_t, int n_threads2){
        sync_t->flag = 0;
        sync_t->count = 0;
        sync_t->n_threads2 = n_threads2;
        int lc = pthread_mutex_init(&(sync_t->lock), NULL);
        assert(lc == 0);
        lc = pthread_cond_init(&(sync_t->cond), NULL);
        assert(lc==0);
}

void th_create(thread_sync *threads, my_arg my_args[], int n_threads, int n_threads2){
        initialize_synch(&(threads->sync_t), n_threads2);
        threads->thread = (pthread_t *)malloc(n_threads*sizeof(pthread_t));
        for(int i=0; i<n_threads; i++){
                my_args[i].sync_t = &(threads->sync_t);
                my_args[i].tid = i;
                 pthread_create((threads->thread + i), NULL, my_args[i].routine, &my_args[i
        }
}

void th_join(thread_sync *threads, int n_threads){
        for(int i=0; i<n_threads; i++){
                 pthread_join(*(threads->thread + i), NULL);
                 printf("Joined thread %d\n", i);
        }
        free(threads->thread);
}
```

The output of the sample main function shown in Listing 4 is shown in the below listing:

```
Hello...you entered thread 0 with argument 0
Now we enter operation zone in thread 0
Hello...you entered thread 1 with argument 1
Now we enter a barrier for thread 1
Hello...you entered thread 5 with argument 25
Now we enter a barrier for thread 5
```

```
Hello...you entered thread 2 with argument 4
Hello...you entered thread 4 with argument 16
Now we enter operation zone in thread 4
Hello...you entered thread 6 with argument 36
Now we enter operation zone in thread 6
Operation complete for thread 4
Hello...you entered thread 7 with argument 49
Operation complete for thread 6
Now we enter operation zone in thread 2
Hello...you entered thread 3 with argument 9
Now we enter a barrier for thread 3
Operation complete for thread 0
Operation complete for thread 2
Joined thread 0
Now we enter a barrier for thread 7
Barrier crossed for thread 1
Barrier crossed for thread 3
Barrier crossed for thread 5
Joined thread 1
Joined thread 2
Joined thread 3
Joined thread 4
Joined thread 5
Joined thread 6
Barrier crossed for thread 7
Joined thread 7
```

We observe that threads in Set 2 i.e. the threads with even IDs are not stopped at any barriers and a joined after their operation gets completed after a certain point. This is especially visible when Thread 0 is joined when Thread 7 has not even encountered the barrier yet. Thus, our selective barrier synchronization primitive is maintaining the correctness of operation while making the synchronization more efficient for the Set 2 of threads(threads with even ID).

```
#include "countdown.h"

void *myfunc1(void *args){
        my_arg *m_args = (my_arg*)(args);
        int tid = m_args->tid;
        int *myfunc_args = (int *)(m_args->args);
        printf("Hello...you entered thread %d with argument %d\n", tid, *myfunc_args);
        printf("Now we enter a barrier for thread %d\n", tid);
        selective_barrier_wait(m_args->sync_t);
        printf("Barrier crossed for thread %d\n", tid);
        return NULL;
}

void *myfunc2(void *args){
        my_arg *m_args = (my_arg*)(args);
        int tid = m_args->tid;
        int *myfunc_args = (int *)(m_args->args);
        printf("Hello...you entered thread %d with argument %d\n", tid, *myfunc_args);
        printf("Now we enter operation zone in thread %d\n", tid);
        for(int i=0; i<20000; ){
                i++;
        }
        printf("Operation complete for thread %d\n", tid);
        selective_barrier_signal(m_args->sync_t);
        return NULL;
}

int main(){
        int n_threads = 8;
        int n_threads2 = 0;
        thread_sync threads;
        my_arg my_args[n_threads];
        int arg_vec[n_threads];
        for(int i=0; i<n_threads; i++){
                arg_vec[i] = i*i;
                my_args[i].args = &arg_vec[i];
                if(i%2){
                        my_args[i].routine = myfunc1;
                }else{
                        my_args[i].routine = myfunc2;
                        n_threads2++;
                }
        }
        th_create(&threads, my_args, n_threads, n_threads2);
        th_join(&threads, n_threads);

        return 0;
}
```

Listing 4: Sample main.c file implementing selective barrier synchronization primitive