OU3(MDU)

Gazi Md Rakibul Hasan
CS: ens20ghn

NOVEMBER 21, 2022

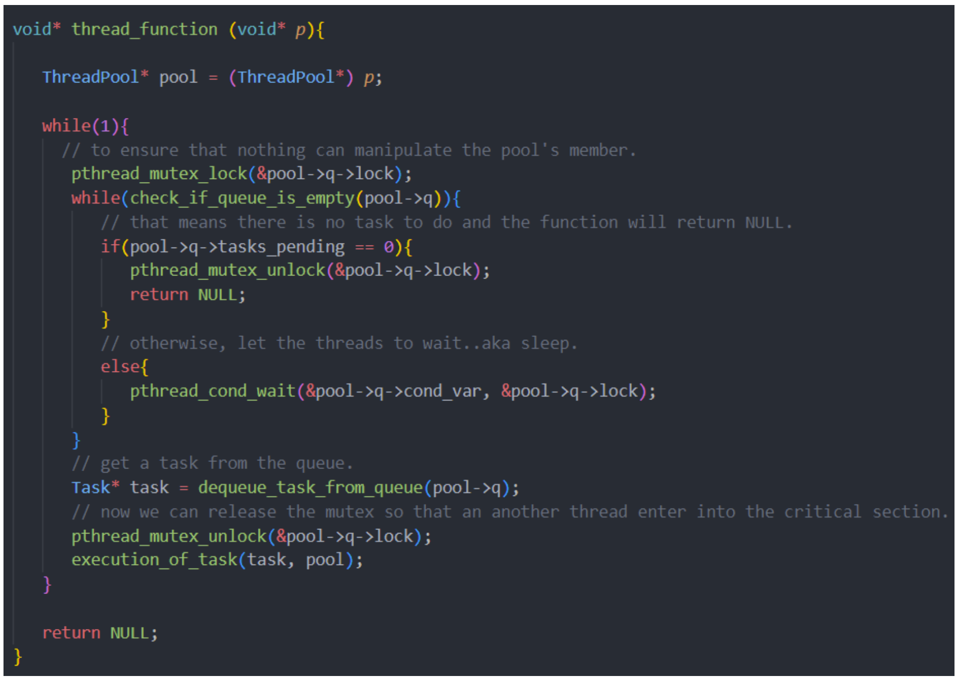
Unix & C programming

Brief Introduction & User Usage

In that assignment, we will implement a program that will determine the size of a directory in the disc of 512 bytes. The output of our program will correspond to the command **du -l -s -B512 [file/directory]** in the terminal. To run the executable file in the terminal, the user should include the file/files or directory/directories along with the executable file **./mdu [files/directories]**. By the following command, the program will run without multiple threads. Suppose the user includes the **-j** flag along with the executable file. In that case, the user must specify the number of threads the program will use to calculate the file's total size or directory in the disc. That can be done by writing **./mdu -j [number of threads] [files/directories]** in the terminal.

Thread Safety of Our Program

Thread safety is essential if we want to run our program in parallel with the help of multiple threads. The definition of thread safety states that the different threads can access shared resources without exposing erroneous output or producing the unpredictable result. Hence, using threads, the program should behave accordingly and achieve the program's specification without unintended interaction[1]. Figure 1 depicts the thread function which we used in our program. That thread function will be given as the argument of **pthread_create().** To achieve thread safety, we must look at the shared resources we have shared in that function. We know that a thread shares the heap memory section and data section with other threads. However, each thread's stack section will be excluded from the other threads. Library functions which are not re-entrant will not be thread-safe.

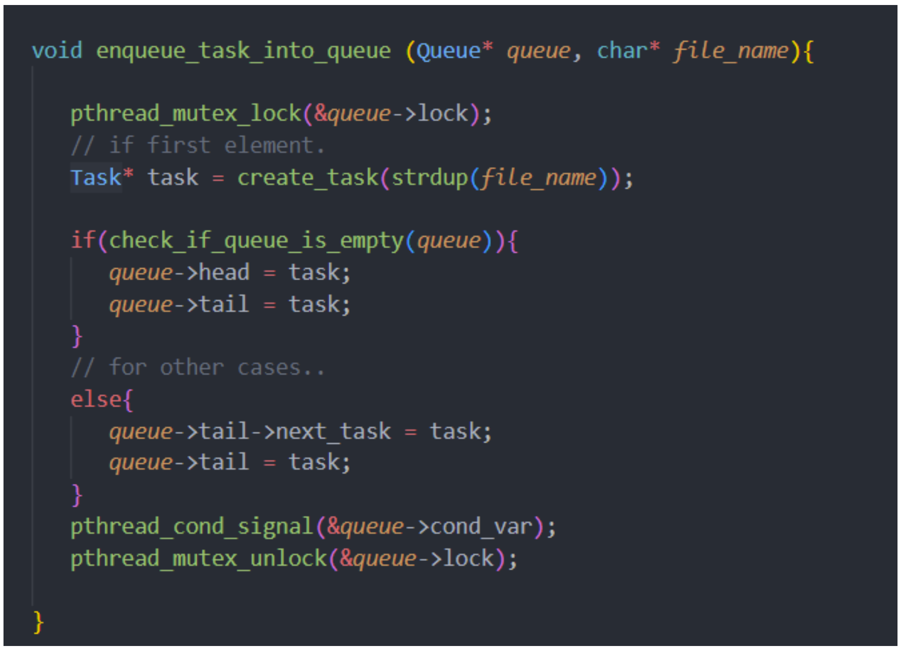


```c
void* thread_function (void* p){

   ThreadPool* pool = (ThreadPool*) p;

   while(1){
     // to ensure that nothing can manipulate the pool's member.
      pthread_mutex_lock(&pool->q->lock);
      while(check_if_queue_is_empty(pool->q)){
         // that means there is no task to do and the function will return NULL.
         if(pool->q->tasks_pending == 0){
            pthread_mutex_unlock(&pool->q->lock);
            return NULL;
         }
         // otherwise, let the threads to wait..aka sleep.
         else{
            pthread_cond_wait(&pool->q->cond_var, &pool->q->lock);
         }
      }
      // get a task from the queue.
      Task* task = dequeue_task_from_queue(pool->q);
      // now we can release the mutex so that an another thread enter into the critical section.
      pthread_mutex_unlock(&pool->q->lock);
      execution_of_task(task, pool);
   }

   return NULL;
}
```

Figure 1: The thread function used as the argument of **pthread_create().**

Figure 1 shows that the mutex protected a critical section of that function at the very beginning of that thread function. It will allow us to take one thread at a time from the thread pool. The queue is dynamically allocated, and allowing many threads to modify the queue simultaneously may cause a thread safety issue. Hence, in the beginning, we will let one thread at a time check if the queue is empty or not. We are then allowing the thread to dequeue a task from the queue. Afterwards, the thread will get a task to execute. After releasing the mutex, another thread will enter into the critical section. It will repeat the same process to get a task from the queue and execute it. If a thread enters that critical section but has no task (when the queue is empty) to do, it will go into the sleep state. In such a way, threads will get a task from the queue without corrupting any thread safety issue. We will get an erroneous result if we don't use the mutex at the critical section. For instance, a thread will try to get a task from the empty queue. After that, it will try to execute it, eventually leading to an erroneous result.

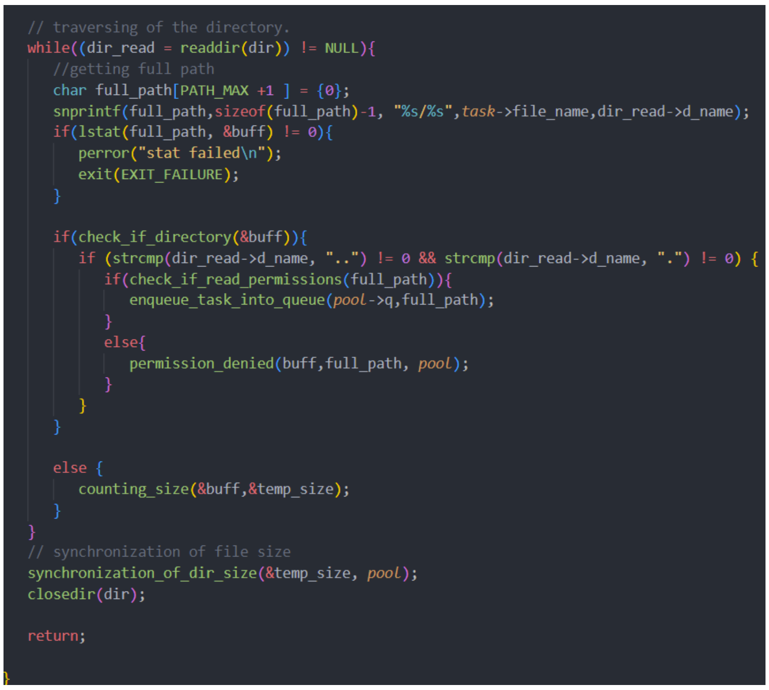
Figure 2 represents the enqueue operation of the queue. We will consider the enqueue operation as a critical section. The traversal of a directory is depicted in Figure 3. Suppose we encounter a subdirectory while traversing the directory. That subdirectory will be added to the queue using the enqueue function.



```
void enqueue_task_into_queue (Queue* queue, char* file_name){

    pthread_mutex_lock(&queue->lock);
    // if first element.
    Task* task = create_task(strdup(file_name));

    if(check_if_queue_is_empty(queue)){
        queue->head = task;
        queue->tail = task;
    }
    // for other cases..
    else{
        queue->tail->next_task = task;
        queue->tail = task;
    }
    pthread_cond_signal(&queue->cond_var);
    pthread_mutex_unlock(&queue->lock);

}
```

Figure 2: That function will be used to enqueue a task into the queue.

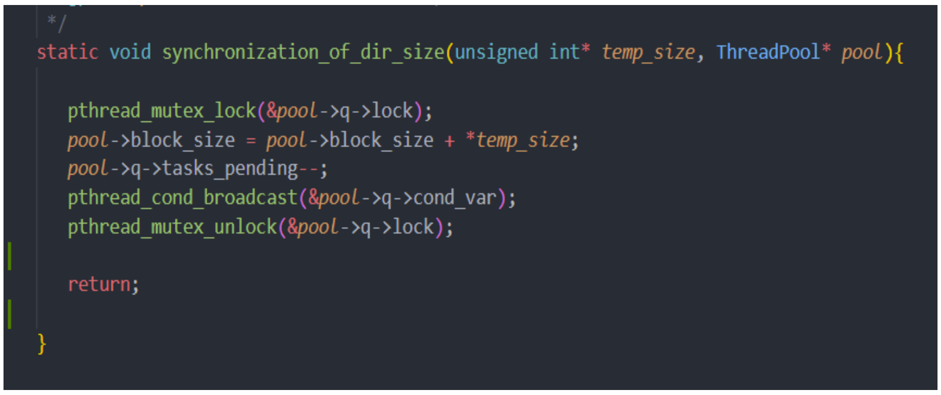
The queue is dynamically allocated in the heap memory section. Thus, it will share its memory with other threads. There will be a risk of possible data race since two threads will want to enqueue a task into the queue simultaneously. Adding protection to the mutex can prevent data races since only one thread can perform the enqueue operation concurrently. Meanwhile, other threads must wait until the mutex releases a thread. In such a way, we will obtain the thread safety for the enqueue operation.



```
// traversing of the directory.
while((dir_read = readdir(dir)) != NULL){
    //getting full path
    char full_path[PATH_MAX +1 ] = {0};
    snprintf(full_path,sizeof(full_path)-1, "%s/%s",task->file_name,dir_read->d_name);
    if(lstat(full_path, &buff) != 0){
        perror("stat failed\n");
        exit(EXIT_FAILURE);
    }

    if(check_if_directory(&buff)){
        if (strcmp(dir_read->d_name, "..") != 0 && strcmp(dir_read->d_name, ".") != 0) {
            if(check_if_read_permissions(full_path)){
                enqueue_task_into_queue(pool->q,full_path);
            }
            else{
                permission_denied(buff,full_path, pool);
            }
        }
    }

    else {
        counting_size(&buff,&temp_size);
    }
}
// synchronization of file size
synchronization_of_dir_size(&temp_size, pool);
closedir(dir);

return;

}
```

Figure 3: This function will be used to traverse the directory.

In Figure 4, we have represented the function that synchronises a directory's size. When a thread has counted a directory's size, that directory's total size will be saved in a local variable. That function in figure 4 will take that local variable's reference as the argument. This function belongs to the critical section since these two variables are shared by all threads. The first variable contains information about the total size of the directory. In contrast, the second variable decrements the number of pending tasks. The absence of mutex protection will create a data race since multiple threads will attempt to modify these shared variables simultaneously. Eventually, it will lead to an erroneous result. Thus, protecting these variables with the help of mutex will ensure thread safety.



```
*/
static void synchronization_of_dir_size(unsigned int* temp_size, ThreadPool* pool){

    pthread_mutex_lock(&pool->q->lock);
    pool->block_size = pool->block_size + *temp_size;
    pool->q->tasks_pending--;
    pthread_cond_broadcast(&pool->q->cond_var);
    pthread_mutex_unlock(&pool->q->lock);

    return;

}
```

Figure 4: That function will synchronise the directory's size.

Analysation of The Program's Performance

In Figure 5, we have depicted the graph that illustrates how much time our program consumes when utilising a specific number of threads. To analyse the performance of our program, we must determine how much time each thread requires to determine the size of the **/pkg/** directory on itchy at Umeå University.

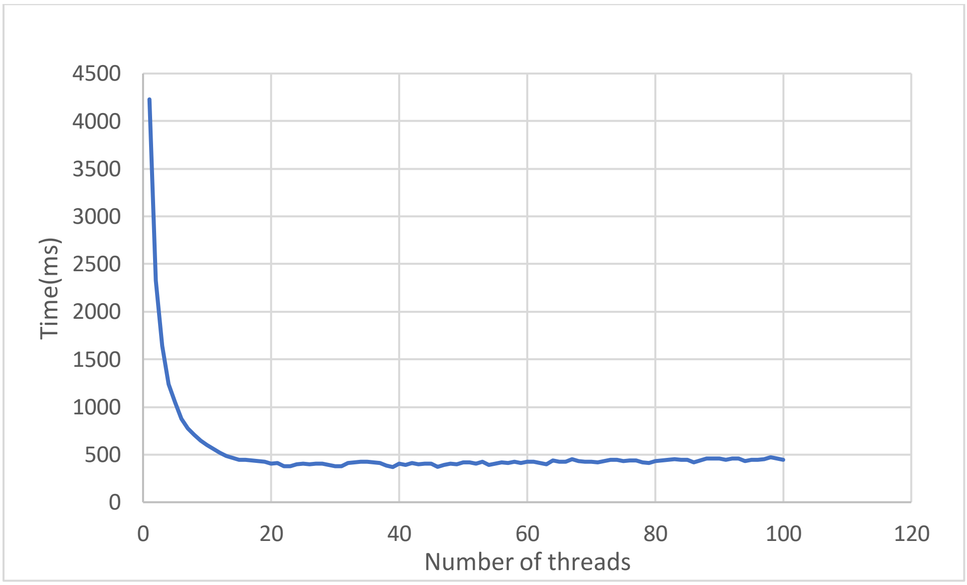


Figure 5: That graph shows the time our program is taking to count the **./pkg/**directory's size with a certain number of threads which starts from 1 to 100.

The Itchy's processor has 64 cores, and its ram is 128 GB. We will use the bash script to obtain performance data. Where we will have a variable n, we will initially assign one to that variable. The current value of n represents the total number of threads that will be executed alongside the executable file in order to determine the size of the**./pkg/** directory. At the same time, we will use the time function to measure the real-time that has been taken to count the given directory size. Then we will update the variable n by incrementing it. We will repeat this process until the variable n becomes 100. In figure 5, we have represented the number of threads in the x-axis, and the y-axis represents the time in microseconds. As we can observe from figure 5, when we will use one thread in our program takes a significant amount of time to determine the given directory's size. However, when the number of threads increases, we can see that our program consumes relatively less time to calculate the given directory size. It behaves in this way because all tasks, including traversal through subdirectories and file and directory size counting, are performed by a single thread. If the number of threads increases, then there will be threads to distribute tasks among other threads. Consequently, it improves the program's ability to determine the size of a given directory within less amount of time.

Nevertheless, we can observe from figure 5 that from the 25 threads to 100 threads, the time our program takes to compute the given directory's size becomes almost constant. It indicates that all given threads will not work simultaneously to determine the size of the given directory. There will be threads in the sleep state. Our program's algorithm has been designed to awaken sleeping threads when a task is in the queue. Thus, increasing the number of threads will not reduce the time required by our program to calculate the directory's size. It may increase the number of sleeping threads instead of active threads.

Discussion

I must admit that this is one of the most difficult assignments I have ever completed. It required considerable effort and time to complete that assignment. However, the most challenging aspect of this assignment was debugging. It is interesting to see how assigning multiple threads to our program can increase its processing power.

Sources:

[1] Wikipedia, Kerrisk, Michael (2010) https://en.wikipedia.org/wiki/Thread_safety