OS HW 3 Dry part

Submitters:

Shaked Or	308026400	shakedor@campus.technion.ac.il
Aviad Rozenkof	316592922	rozen501@gmail.com

Q1:

1. Let as label a locked list of size n as L and the elements of said lists as {ai | 1≤ i ≤ n}.
   Note that each element has a mutex lock in its node.
   Searching:
   We lock a1 and then look at the data of a1, if a1 does not contain the data desired we lock a2 unlock a1 (after which only a2 is locked by us) and then look at the data of a2.
   We continue to do so until the end of the list or until our element if found. If found we keep the lock only on the desired element and if endOfList is reached we unlock the last node before declaring failure. The idea is that we lock ai and a(i+1) when moving from the former to the latter so that the connection between them cannot be changed while we attempt to move between them.
   Inserting: suppose we want to insert aj between ai and ak=a(i+1). We lock aj and ak in that order to avoid deadlocking (unless the convention of the program is to iterate backwards). We then create and lock aj (not necessary if no other thread knows aj) and link it to ai and ak (note that during the linking all 3 are locked by our thread). We then unlock all 3 nodes in any order.
   Removing: suppose we want to remove aj from in between ai and ak (i=j-1, k=j+1).
   We acquire the locks of ai , aj and ak in that order to avoid deadlocking. Then we delink aj from ai and ak and link ai and ak to each other. Then we can deallocate aj. Afterwards, we unlock, the now linked, ai and ak in any order we choose.
   Note that when inserting and removing element to and from the 2 ends of the list (meaning first and last) we do not have a previous/next element to lock and thus we only lock and link the subset of ({Lneighbor, target, Rneighbor}) that exist in the list.

2. Observation 1:
   Let there be T1, p1, T2 and p2 as given in Observation 1 (denoted obs1). Since T2's candidate, p2, is bigger than p1, this means that T2 tried to select p1 as a candidate before but upon not finding p1^2 realized that another thread is handling p1. This means that at that point in time, T1 was ahead of T2 in the list (since it was handling p1 before T2). Then T2 went back to p1 and tried getting another candidate and eventually reached p2. Since we assume that T1 had not finished removing elements that are divisible by p1, and we know that a thread cannot overtake another thread that is ahead of it in the list (unless the leading thread goes back to choose a new candidate) then T2 is necessarily behind T1 in the list in the scenario described by obs1. Thus the element locked by T2 is smaller than the element locked by T1.
   Observation 2:
   Reminder, a thread T with a given candidate p can only remove elements that are divisible by p. Let p be a prime candidate. By ob3 (explained later) p is prime. Thus p^2 is only divisible by 1, p and it self. 1 is not in the list and thus is never a candidate. If p^2 is a candidate it is a prime by obs3 which contradict the fact the it is the square of a prime. Thus the only possible thread that can remove it is T with given candidate p.
   Observation 3:
   Let there be p a prime candidate of thread T. if T managed to reach the state that it chose p as a candidate, that means that for all k<p either k was removed by a Thread prior to this moment (possibly by T itself) or k was a candidate that T must have considered, meaning it either chose k

as a candidate and removed all of the elements corresponding to k or it reached k^2 and realized that another thread was handling it.
Let as assume that p is not prime. Thus it has a minimal prime component, e. since e is prime and is smaller than p, T must have considered e as a candidate. This lives us with 3 cases:

- T handled e
- Another thread T' handled e
- Another thread T' is currently handling e

From the first 2 and the fact that p is divisible by e we get a contradiction since p was removed from the list and yet was selected as a candidate.
From the 3rd and obs1 we can deduce that the element locked by T', e', is bigger than p. meaning that T' already passed p but since e divides p then p was removed. Contradiction.
Thus p is a prime candidate.

3. If all threads run the unmodified algorithm, then a thread T will reach candidate p only after reaching the end of the list trying to remove divisors of k for every k, a candidate smaller then p.
   Now, let T be the thread that selected 2 as a candidate first and T' any other thread.
   We will show by induction that T' could not have removed any element.
   Since all threads run the unmodified algo, then T' will only choose 3 as a candidate after it had reached the end of the list trying to find elements divisible by 2. Note that it won't find a single element divisible by 2 since it cannot overtake T and T is removing all of those elements. But if T' reached the end of the list it means that T already reached there and chose 3 as its candidate first. Meaning, like the case with 2, that T will remove all divisors of 3 that remain and T' will remove nothing. This goes on by induction for every candidate. Thus, the thread that reached 2 first will remove all elements and the other threads will do no real work.

Q2:

1. All nodes in the list contain a read write lock. All threads use hand over hand read/write locking. The threads will use read locks to search the list and acquire write locks when trying to remove elements from the list. When a thread sees a candidate is already being read it assumes another thread is handling that candidate and moves on to the next candidate. Under this method let as imagine the following scenario: T1 chooses p1 as a candidate, while T1 is reading it, T2 reads it and sees it is already red. T2 assumes another thread is handling p1 and moves on to a bigger candidate p2 and remains ahead of T1 for the duration of the current candidate iteration.
   In this case we see that T2 has a bigger candidate and is ahead of T1 in the list.
2. In the method explained above, let us take the following scenario: T1 chooses 2 as a candidate and is reading it. T2 sees 2 is taken, moves on to 3 and is reading it. Then T3 comes, while both T1 and T2 are reading their respective candidates, and then moves on to take 4 as a candidate. Since 4 is not prime, we see a scenario where a non-prime is chosen as a candidate (although it will not remain in the list after the algorithm finishes running).

Q3:

1. First let us look at what It means to be able to change from a reader to a writer atomically. In order to be a reader and change to a writer without waiting after unlocking read and before locking write we must be the only readers and have no other writers (otherwise we risk data corruption). Thus what the heuristic of the upgrade from read to write function is: Wait till I'm the last reader, and then change to writer atomically.
   With this heuristic the problem is more easily identified. If 2 threads are allowed to upgrade from read to write, a scenario were both are readers and are waiting to be able to upgrade can occur. In this scenario both threads are waiting till they are the only readers remaining. They will wait indefinitely and we are going to have a deadlock.

2. Code given Below

```c
#include <pthread.h>

int number_of_readers;
pthread_cond_t readers_condition;
int number_of_writers;
pthread_cond_t writers_condition;
mutex_t global_lock;
// bool says if a thread was selected to become the only one that can upgrade.
// upgradable stores the thread id of the thread that is upgradable.
// we use bool instead of looking for a NULL like initial value for upgradeable.
pthread_t upgradeAble;
bool upgraded;
// this condition checks when we can upgrade
pthread_cond_t R_to_W_condition;

void readers_writers_init() {
	number_of_readers = 0;
	pthread_cond_init(&readers_condition, NULL);
	number_of_writers = 0;
	pthread_cond_init(&writers_condition, NULL);
	pthread_mutex_init(&global_lock, NULL);
	upgraded = false;

}

bool upgrade_to_write_lock()
{
	pthread_mutex_lock(&global_lock);
	//if no upgraAble was chosen, choose it
	if (!upgraded){
		upgradeAble = pthread_self();
		upgraded = true;
	}
	//if the current thread is not upgradable, return 0;
	if (!pthread_equal(upgradeAble, pthread_self())){
		pthread_mutex_unlock(&global_lock);
		return false;
	}

	if (number_of_writers > 0){
		// cant be a reader since there is a writer active so either we are non
readers or are the writer itself
		return false;
	}

	while (number_of_readers != 1 && number_of_writers != 0){
		pthread_cond_wait(&R_to_W_condition, &global_lock);
	}

	number_of_writers++;
	number_of_readers--;
	pthread_mutex_unlock(&global_lock);
	return true;


}
```

```c
void read_lock() {
	pthread_mutex_lock(&global_lock);
	while (number_of_writers > 0)
		pthread_cond_wait(&readers_condition, &global_lock);
	number_of_readers++;
	pthread_mutex_unlock(&global_lock);
}


void read_unlock() {
	pthread_mutex_lock(&global_lock);
	number_of_readers--;
	if (number_of_readers == 0)
		pthread_cond_signal(&writers_condition);
	//might need to signal to upgrade if only 1 reader
	else if (number_of_readers == 1)
		pthread_cond_signal(&R_to_W_condition);

	pthread_mutex_unlock(&global_lock);
}

void write_lock() {
	pthread_mutex_lock(&global_lock);
	while ((number_of_writers > 0) || (number_of_readers > 0))
		pthread_cond_wait(&writers_condition, &global_lock);
	number_of_writers++;
	pthread_mutex_unlock(&global_lock);
}

void write_unlock() {
	pthread_mutex_lock(&global_lock);
	number_of_writers--;
	if (number_of_writers == 0) {
		if (number_of_readers == 1)
			pthread_cond_signal(&R_to_W_condition);
		pthread_cond_broadcast(&readers_condition);
		pthread_cond_signal(&writers_condition);
	}
	pthread_mutex_unlock(&global_lock);
}
```