Homework 3 Wet

Due date: Monday, 4/06/2012 12:30 noon

Teaching assistants in charge:

- Anastasia Braginsky

All emails regarding this assignment should be sent only to cs234120@cs.technion.ac.il and must have the subject **cs234123hw3_wet.**

Note: Start working on the assignment as soon as possible!!! This assignment involves algorithmic design, a lot of code writing, and extensive testing. **There will be no postponement.**

1. Introduction

Welcome to the world of concurrent data structures! ☺

You are to implement a program which enables both updates and queries to a doubly linked list, of the same kind learnt in the Data Structures course, concurrently. Fine grained locking (as explained in the class) will be used to accomplish concurrency. The operations (updates and queries) are given as a series of bursts of commands from the input, so that all commands in a burst are to be executed parallel, and every burst must end before the next burst begins. For the doubly-linked list implementation, you will also implement a lock mechanism called Read-MayWrite-Write lock. In order to send bursts of commands to the list, you will implement a mechanism called Barrier.

Note that this exercise (in contrast to the previous two) is going to be implemented in the User Mode (no kernel changes and compilations).

2. General Description

Doubly Linked List

A doubly linked list is a fundamental data structure and is used in kernel as well as user level applications. You will implement an ordered doubly linked list sorted by key, with HEAD pointing to the end of the list with low-value keys and with TAIL pointing to the end of the list with high-value keys. Every node carries an additional data field that is unique for every key, and is kept in the node. This data is a single character. For initialization issues, you can assume that key 0 and data '0' is never in use. Your doubly linked list should support the following operations:

`void Initialize ()`

Name	Initialize
Description	Initializes the global doubly linked list structure. The function should not be concurrent.
Input Parameters	None
Output Parameters	None
Comments	This function should be called once, before any concurrent access to the doubly linked list structure is allowed.

`void Destroy ()`

Name	Destroy
Description	Destroys and frees the global doubly linked list structure. The function should not be concurrent.
Input Parameters	None
Output Parameters	None
Comments	Since function is not concurrent it should be called once, when any concurrent access to the doubly linked list structure is no longer possible.

`Bool InsertHead (int key, char data)`

Name	InsertHead
Description	Inserts the key, with the appropriate data, into the ordered doubly linked list. The search for the appropriate location in the list must start from the head of the list.
Input Parameters	The key and its associated data
Output Parameters	FALSE – if the key was found as already existing in the list, TRUE otherwise.
Comments	You can assume the input parameters are valid

`Bool InsertTail (int key, char data)`

Name	InsertTail
Description	Inserts the key, with the appropriate data, into the ordered doubly linked list. The search for the appropriate location in the list must start from the tail of the list.
Input Parameters	The key and its associated data
Output Parameters	FALSE – if the key was found as already existing in the list, TRUE otherwise.
Comments	You can assume the input parameters are valid. InsertHead and InsertTail are both searching for the appropriate place, but in different direction, and them both should have exactly the same outcome.

`Bool Delete (int key)`

Name	Delete
Description	Deletes the key, with the appropriate data, from the ordered doubly linked list. The search for the appropriate location in the list should start from the head of the list.
Input Parameters	The key
Output Parameters	FALSE – if the key wasn't found as already existing in the list, TRUE otherwise.
Comments	You can assume the input parameter is valid.

`Bool Search (int key, char* data)`

Name	Search
Description	Determines whether the key exists in the ordered doubly linked list. The search for the appropriate location in the list should start from the head of the list. If the key is found, its associated data is returned.
Input Parameters	The key and the pointer to the place where the associated data should be written in case of success.
Output Parameters	FALSE – if the key wasn't found as already existing in the list, otherwise TRUE and the char referenced by the pointer should be updated to the appropriate data.
Comments	You can assume the input parameters are valid.

The deletion and search in the list are the same as you learned in the DS course. The inserts are also the same just differ in the start points. The list should not support multiple keys, and insertions of already-existing keys, even with different values, are not allowed (i.e., there are no overwrites).

The complexity of the Initialize() operation should be O(1). The complexity of each of the remaining operation should be O(*k*), where *k* is the number of nodes that need to be passed according to the value of the key. In other words, in order to implement the InsertTail() operation you can not start search from the tail, reach the head, and then do the InsertHead() operation.

Executing Commands

You receive, through the standard input, several bursts of commands. Every burst contains the following possible commands:

1. Inserting a pair (key, value) to the list (from the head or from the tail). This insertion should only succeed when the key is **not** already in the list. This command should return a true/false value according to its success. The values can be any single character (and are kept as such). Keys are integers.
2. Deleting a specific key together with its value. The deletion only succeeds if the key exists.
3. Finding the value for a specific key. This command should return the value, if the key exists, or a failure value, if it doesn't.

You should read every burst from the input as a whole, and then try and execute all of its commands simultaneously. For this matter, you should assign a new thread for each task, and destroy the threads upon completion of all commands in the burst. Notice that you must let the threads work simultaneously. Meaning, you should neither let the threads run one after another, nor let any of the threads breach the boundary of the burst. All threads should work in parallel inside the boundaries of a single burst and terminate before the next burst is read. (At which point, new threads will be allocated for the new burst).

In general ***Barrier*** synchronization structure is initialized to *n* - the number of the threads it should wait for. Then each of the threads that should have been synchronized on this barrier should call *barrier()* method, which is blocking till the time when *n* threads reach the barrier (call *barrier()*). Then all *n* threads should be simultaneously allowed to run and can continue running in any order according to the scheduler. After that the barrier can be reinitialized to any value.

The design should comply with the following guidelines:

- There should be a one thread (called main thread) that should read the burst from the standard input.
- The main thread creates the needed amount of the threads (workers) that will process the commands.
- The main thread initiates the barrier and supply input to the threads in the burst (for example if the burst contains 7 commands then main thread need to prepare 7 threads).
- The worker threads wait till all the workers have been created (on barrier) and only after it they start processing the command. For this you have to design a mechanism, called Barrier, explained above.
- After the command has been finished each worker prints the output to the standard output and ends.
- When all the worker threads has finished the main thread can start reading the next burst.

You should devise a synchronization mechanism that ensures the validity of the list with most possible concurrency. **Notice that using a single lock for the list is not allowed – such a HW will receive 0 points automatically!** Namely, the rules are that any number of threads may enter and read a node to which no one writes to, but a thread that writes to a node must be exclusive to it, and block other attempts to read it. Additionally you have to synchronize the input and the output as explained above and below.

As an effect, your output will not always be deterministic. Suppose you have a burst with the following commands:

(a) InsertHead (3,a)
(b) InsertTail (3,b)
(c) Search(3)

Then it is possible that either (a) gets the lead, and then (c) should return "a", or that (b) gets the lead, and then (c) should return "b". Finally, (c) may get the lead, in which case it should return "failure". Notice that in any case, either (a) or (b) should succeed, and that should the (c) command appear in the next burst, it would succeed as well (and return either "a" or "b").

3. Detailed description

Your program will consist of a number of threads:

1. The main thread, which reads the bursts from the input, and allocates the threads for each command in a burst, and waits upon their termination before it reads another burst.

2. The threads allocated for their sole commands.

The threads must follow these rules:

- No communication is allowed between the main thread and the threads other than initialization, and notification of completion.
- Only the main thread should access the standard input.
- The threads must not communicate with each other.

Synchronization Mechanism

In your synchronization you have to use new kind of the Read-Write lock, which is called a **"Read-MayWrite-Write" lock**. It has additional features to the Read-Write lock that you have learned in class. The motivation for the Read-Write lock is that there can be multiple reader threads which can be executed together. The novelty of the "Read-MayWrite-Write" lock is that it has a new "MayWrite" mode, for the possibility that the thread still doesn't know whether it would just read the data of that node, or would also write the data. Therefore, it is possible to acquire the lock in MayWrite mode, which, at this point, allows performing reading only. Other readers are allowed to enter the lock when the thread is in the MayWrite mode. If, at the end, the thread should not write anything in this node, it will just release the MayWrite lock (as if it were a reader all along). However, if that thread needs to

write data in the node, it requests an upgrade on the may-write lock to become a writer. After requesting an upgrade, the thread waits for all the readers that share the lock to exit the critical section (i.e. release the locks), and doesn't allow new readers to enter. When the upgrade request returns, the thread upgrades to a writer mode, (which is exclusive in this lock), and can write data!

The new lock should support the following interface (assuming your new lock is defined in structure *lock*):

- **`get_read_lock(lock* l)`** - *Get lock in the read mode*: Allowed if there are currently just readers or one may-writer in addition to the readers, which didn't request an upgrade. Notice that as opposed to what you have seen in the class, **we want this lock to be fair for the writers and the readers!** Thus, if there is a writer waiting for the lock we do not allow the reader to obtain it. Also, if there is a may-writer waiting for the upgrade on this lock we also do not allow the reader to obtain it.

- **`get_may_write_lock(lock* l)`** - *Get lock in the may-write mode:* Allowed if there are just readers in the lock, and no additional may-writers. Also there should be no writers or updater already waiting. We allow only one may-writer at a time because we do not want to deal with two or more simultaneous upgrade requests.

- **`get_write_lock(lock* l)`** - *Get lock in the write mode:* Allowed if there is no other thread in the lock. Notice that now it is important to remember that there is a writer waiting.

- **`upgrade_may_write_lock(lock* l)`** - *Upgrade lock in may-write mode:* Notice that this is relevant only if the lock is in MayWrite mode. You may use some assert here. There is no upgrade for either readers or writers! Also notice that this upgrade has a priority over any waiting writer! Any waiting writer will have to continue to wait till this may-writer (that is turning to be writer) will receive an upgrade, and be done. The thread waiting for an upgrade will, in turn, need to wait till all current readers are done. The upgraded thread becomes a full writer for all purposes (no implications from formerly being a MayWriter).

- **`release_shared_lock(lock* l)`** - *Release lock that was taken in shared mode:* should be called by the thread holding the lock in the read mode or in the may-write mode that wasn't upgraded.

- **`release_exclusive_lock(lock* l)`** - *Release lock that was taken in exclusive mode:* should be called by the thread holding the lock in the write mode or by the upgraded may-write mode.

Notice that a "may-writer" thread does not lose its place at the head of the lock's waiting queue, when requesting an upgrade, and that this cannot be implemented using the regular readers-writers lock. You should devise a method, using the synchronization methods available in pthreads, to implement this lock. Note that the waiting queues included in all synchronization primitives supplied by pthreads (mutex, semaphore, conditional variables) are not fair! Those waiting queues are FIFO by priority, but we require this new lock to be fair (FIFO according to the arrival time) regardless the priority.

Notice that whatever structure and algorithm you will use for R-MW-W locking you will need to cover it with another internal lock. So, only for this internal synchronization, do not worry that original order of the threads can be changed due to waiting for this internal locking. You can assume that the threads are arriving to your R-MW-W lock in the order as they passed the internal lock. While after that you need to keep their arrival order.

Such a lock must be assigned to every node, and limits the accessibility to enter this node. Carefully consider the implementation of the synchronization on the list, for queries use only read locks and for insertions and deletions use combination of may-write and write locks. Your operations should be implemented with maximum concurrency. Also notice that the order of entering the locks on the doubly linked list is very important! You can easily get into the deadlock if you use an opposite order!

More explanations about fairness:
We want the lock to be fair according to the time of arrival of the threads. A thread waiting to get the write-lock for 10 minutes can not be bypassed by another writer-thread that has just arrived. Similar should not happen for any other lock mode. Finally, if some threads truly arrive simultaneously (very close one to the other) at the beginning of their lock requests, it is OK to swap their order.

As an example (see Figure 1), suppose that there are five threads that request the lock on a node A. Threads 1 and 2 first request an R permission, and get it. Thread 3 arrives later and requests a MW permission, and gets it (because only readers are in node A). Then, thread 4 requests a R permission, and also gets it (because thread 3 does not prevent extra readers). Then, thread 3 requests an upgrade on node A. Thread 3 is blocked, and waiting for threads 1, 2, 4 to clear the R permission. Later, thread 5 tries to obtain an R permission on node A, but is blocked in waiting because of the upgrade request by thread 3. Thread 3 obtains a W permission after threads 1,3,4 finished. When thread 3 clears the W permission, thread 5 is free to obtain its requested R permission.

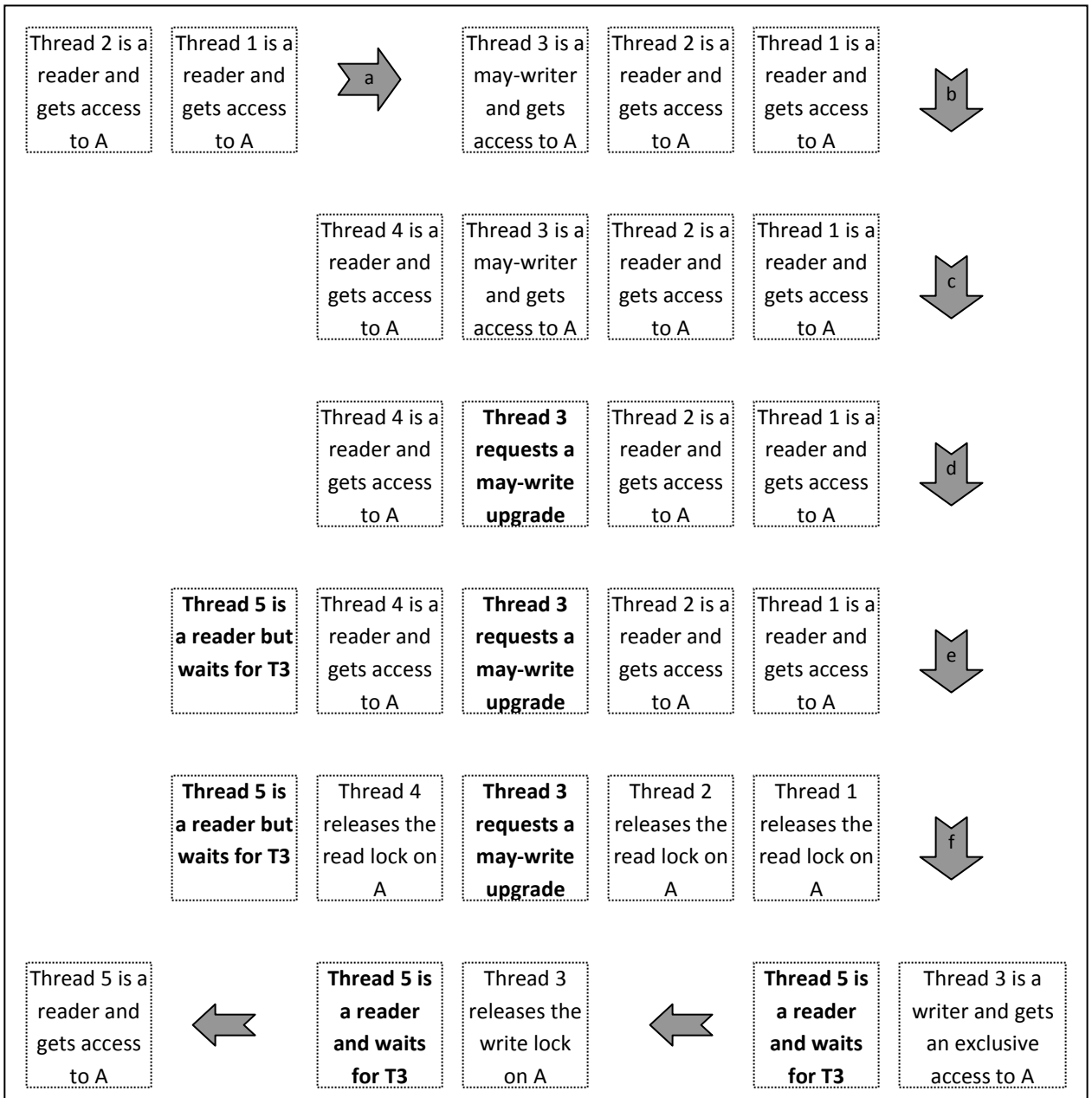

Figure 1: The process of requiring lock on Node A in a timeline. Note that due to the fairness that we require, at Step "f", thread 5 can not bypass thread 3.

Note in general, when using the pthread wait-queue Q, you can not assume thread A will be de-queued from Q before thread B, if A is en-queued to Q before B. This is because the pthread's wait-queues are not fair.

Format of Input & Output

The input will be as follows:

BEGIN
COMMAND 1
COMMAND 2
….
BARRIER
COMMAND 1
COMMAND 2
…..
BARRIER
END

The input always begins with "BEGIN" and ends with "END". The "BARRIER" commands separate the different bursts. Between "BARRIER" commands, all commands must be executed simultaneously, and the bursts must be executed sequentially. The input lines are separated with "\n" characters. The size of each burst doesn't have to be equal.

When each command is executed, a line is written to the output according to the result. The line always consists of the original command (that includes the "BARRIER" command), and the result in the following manner:

Input: COMMAND 1
Output: COMMAND 1->RESULT

The possible commands and their return values are formatted as follows:

1. "INSERT_HEAD #key #value" – Inserting a (key, value) pair to the list from the head. In case of success, the result is TRUE, otherwise, it is FALSE.
2. "INSERT_TAIL #key #value" – Inserting a (key, value) pair to the list from the tail. In case of success, the result is TRUE, otherwise, it is FALSE.
3. "DELETE #key" – Deleting a key from the list. In case of success, the result is TRUE, otherwise, it is FALSE.
4. "SEARCH #key"– searching and returning the value of a key. If the key exists, the result is the value. Otherwise, the result is FALSE.

An example of a possible output:

Input	**Output**
BEGIN	BEGIN
INSERT_HEAD 3 a	INSERT_HEAD 3 a->FALSE
INSERT_TAIL 5 b	INSERT_HEAD 3 c->TRUE
INSERT_HEAD 3 c	INSERT_TAIL 5 b->TRUE
SEARCH 3	SEARCH 3->FALSE
BARRIER	BARRIER
INSERT_TAIL 5 u	SEARCH 5->b
DELETE 5	INSERT_TAIL 5 u->FALSE
SEARCH 3	DELETE 5->TRUE
SEARCH 5	SEARCH 3->c
END	END

Notice that you have to synchronize the output! When, for example, two threads are done and want to print the "SEARCH 3->c" and "SEARCH 5->b" they have to be able to print them separately without any mixture of characters. Of course, the order of the output might be different from the real order of the input inside any burst.

Note that when writing the output line you should add no spaces, other then those you read as part of the command.

4. Remarks

- The assignment should be implemented in C and should work in the VMWare running Red Hat Linux 8.0.
- You should use only pthread library to work with threads.
- Carefully design the node locks so it allows the best parallelism possible, and prevent deadlocks.
- **If your implementation becomes deadlocked, then the penalty in the grade for the whole assignment will be 30 points!**
- You have to submit a detailed description of both of your insertions, deletion and search algorithm. Make sure that they allow maximum parallelism otherwise you will lose points. In addition, don't forget to write a description of the Barrier and “Read-MayWrite-Write" lock implementations and the description of the whole system (pay special attention to simultaneous command execution mechanism and the output printing).
- The suggested work plan: start with creating a design! Think how you will implement everything before starting the coding! Later continue with implementation of the barrier mechanism, then the lock and only after testing these things start implementing the main thread (parsing of the input). And finally the list.
- We also strongly suggest you to use asserts and/or defensive checks for easy debugging. Pay attention that when you submit the hw in your Makefile you will not compile the program for debugging, thus asserts will not slow down the execution. On the other hand it is a very powerful tool for debugging the concurrent programs.
- Don't print anything except the required output.

5. Submission

- You should electronically submit a zip file that contains the source files and the makefile. Its name should be "Makefile". The makefile will create an executable with name "**dbl_list**". Note that this makefile should compile your whole code and not only the dispatcher.

- You should submit a printed detailed design of your program, including explanations on the chosen algorithms, synchronization mechanisms, etc…

- You should also submit a printed version of the source code.

    - A file named **submitters.txt** which includes the ID, name and email of the participating students. The following format should be used:

        Bill Gates bill@t2.technion.ac.il 123456789

        Linus Torvalds linus@gmail.com 234567890

        Steve Jobs jobs@os_is_best.com 345678901

Important Note: Make the outlined zip structure **exactly**. In particular, the zip should contain only the following files (no subdirectories):

```
zipfile  -+
          |
          +- all your source/header files
          |
          +- Makefile
          |
          +- submitters.txt
          |
          +- documentation.pdf
```

בהצלחה,

צוות הקורס