Explaination.

***How we use POSIX Threads and sync primitives to provide maximum concurrency?***

We used pthread library of Linux kernel. We created mutex of type “mutex with error check”. We create 4 different locks which use this mutex.

We achieve max concurrency by not locking the whole list, but each node separately. We lock just a node which we are iterating over, and his neighbor nodes (to prevent deadlocks, (explanation further). So for example we could provide insertions and deletions of different (not neighbor) nodes.

In addition we use two types of locks: for readers and for writers. That allowing a number of readers to have an simultaneous access to nodes, but denies it for writers.

Just in case of list\_free function we are locking the whole list (by blocking head pointer) in order to refuse further operations on a list that is in destroy process.

In addition we don’t just lock each node, but we lock separating fields: in list we are locking head pointer and size separately and in each node we lock the pointers to neighbor nodes, index and data separately. So for example we allow insertion between index i and i+1 and data update in node i simultaneously.

We also use condition variable to allow readers to enter the mutex when writers finished and vice versa. And to implement the starvation counter which prevents starvation of writers (explained further).

In function list\_batch() we created new thread for each task with help of the library function pthread\_create(). After that we are waiting for started threads to finish with help of pthread\_join() for each thread.

***Why no deadlocks?***

We designed our list in such way that it is never allowed to enter into a deadlock situation.. We are violating the “circular wait” condition. Our list is an ordered sequence of nodes (we consider the list struct as a resource of first order and then coming nodes in their order). Every thread can request to lock only subsequent resources and never any previous resources. So violating this rule eliminates the possibility of a deadlock.

***Why no starvation?***

Our implementation avoids writers’ starvation by using a counter which limits the number of active readers in case that there is a writer waiting. After running a maximum of allowed readers, the writer must run, so there won’t be any starvation of writers in case of a scenario that just new readers are coming.

If the number of active readers has reached its limit, then all readers that come after will wait. When all the active readers are done than we promise that the waiting writer will start and not starve. In other words if there is a number of writers (W) and many new readers (r r ... r) waiting so our algorithm will run: W r r ...r W r r … r W r r … r and etc. when the series of readers will not exceed the allowed limit.