**Report (Prog07)**

2020.12.18

Baheem Ferrell, Marta Vargas

1. **Data structure and Implementation**
   1. **Version 1: Single-traveler version, no sliding partitions**

**Data structure**

The number of moves before tail growth is stored in a global integer variable *numMovesForGrowth*.

Due to compilation errors using *g++*, enumeration types and struct types definitions are updated without using *using* keyword in *dataTypes.h*.

At each rendering step, the main thread performs below actions repeatedly.

3 variables are added to Traveler struct to store thread ID (*pid*), number of moves made so far (*moves*) and whether or not the traveler can continue moving (stuck) to be used in each traveler thread.

**Implementation**

*drawTravelers*() is called by a GLUT rendering callback.

Here, only those travelers that have *pid* greater than 0 are rendered.

Command line arguments are parsed into *numRows*, *numCols*, *numTravelers* and *numMovesForGrowth*.

When creating travelers in *initializeApplication*(), *index*, *moves* and *stuck* variables are initialized to the traveler’s index number, 0 and false respectively. After all travelers are created, a traveler thread is created for each traveler using *pthread\_create*() with its index as a parameter to *travelerFunc*().

In *travelerFunc*(), the index is obtained from the input parameter and the corresponding *Traveler* data is obtained from *travelerList*. It executes a *while*-*loop* until the traveler reaches the exit. At every iteration, it first obtains the position and direction of its header block and tries to move it along a random direction excluding the opposite direction. In some cases, all 4 directions are not available due to various reasons (outside the grid, neither empty nor exit). In order to accommodate these cases, an integer variable, *checkedDirection*, is defined and maintained to keep the record of already checked directions. If this value indicates that all 4 directions are already checked, there is no need to try further and the thread function simply exits. Just before it exits, it sets *stuck* to true and decrement *numLiveThreads* by 1.

If a valid direction is found, it checks if the number of moves after tail growth has reached the value given in the argument. If it has, a free block is turned into a traveler block and prepended to *segmentList*. (append a new block, shift elements one by one towards the end, and update the first element with the new grid position and direction) Otherwise, the elements in *segmentList* are updated. (shift elements one by one towards the end and update the first element with the new grid position and direction) In either case, the new grid position is marked as a *TRAVELER* block unless it is EXIT and the number of moves is incremented by 1.

This repeats until either the traveler reaches the exit or it cannot move any more.

Version 1 was tested using the following command line arguments.

*./traveler 40 45 1 10*

* 1. **Version 2: Multithreaded, with no synchronization**

Version 2 is almost the same as Version 1. The only difference is that Version 2 does not check whether the current traveler can move further or not.

This is because in some situation, an already stuck traveler can start moving again in a multi-traveler environment if more than one directions previously unavailable become available again by other travelers moving to different part of the grid. This feature allowed the traveler thread to exit when it is stuck in Version 1, but it is no longer reliable in a multi-traveler environment and hence removed.

Version 2 was tested using the following command line arguments.

*./traveler 40 45 1 10*

* 1. **Version 3: Multithreaded, with a single lock**

In Version 3, a global mutex lock is initialized first. It is then acquired/released by traveler threads and rendering thread to synchronously access the grid and the travelers’ data.

A pair of *pthread\_mutex\_lock*() and *pthread\_mutex\_unlock*() is called at the beginning and at the end of critical sections where the grid and the travelers’ data are read or written.

Essentially, every statements or blocks that access global variables shared among different threads are considered to be critical sections and hence synchronized with the global mutex lock. Care was taken to *break* and *continue* statements within loops for the acquired lock to be released.

* 1. **Version 4: Enable partition sliding and multiple locks**

In Version 4, in addition to a global mutex lock an array of locks is initialized for every traveler. In critical sections (explained in Section 1.3.) access to travelers’ data are synchronized using appropriate mutex lock for that traveler. Note that these blocks are always wrapped by global mutex synchronization to ensure that no deadlock occurs.

Partition sliding is similar to traveler’s movement, except that vertical partitions can slide along vertical axis only (either top or bottom) and horizontal partitions along horizontal axis only (either left or right.

During test, it was observed that some partitions would move back and forth between stuck travelers as travelers keep trying to make way by moving partitions they encounter.

* 1. **Version 5: Synchronization with multiple gridlocks**

In Version 5, a 2-dimensional array of locks is initialized for every block in the grid. In critical sections (explained in Section 1.3.) access to any grid block is synchronized using appropriate mutex lock for that block. Note that these blocks are always wrapped by global mutex synchronization (and traveler’s mutex synchronization if necessary) to ensure that no deadlock occurs. Extra care was taken for the synchronization of partition sliding which involves synchronizing 3 probably different blocks (current head of the traveler, to-be-free block and to-be-partition block due to the sliding) since acquiring the same mutex lock leads to deadlock of that block and hence the global lock.

1. **Data structure and Implementation**

In this project, threads that correspond to already exited travelers are not properly cleared. In addition, thread that correspond to stuck travelers are left running as there is no obvious way to resolve the cause of this behavior without looking into nearby travelers and partitions recursively, which overcomplicates the overall implementation.