Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Matthias Schubert
Felix Borutta

Munich, May 22, 2019

Managing Massive Multiplayer Online Games

SS 2019

Exercise Sheet 4: Persistence

The assignments are due May 29, 2019

Assignment 4-1 *Logging with simple algorithms*

Consider an abstract game with its information being stored server sided. Assume the data to be stored within the objects O_1, O_2 and O_3. Initially, every object O_i contains the value o_i. This means the initial state of the database is as follows:

Object	Value
O_1	o_1
O_2	o_2
O_3	o_3

Starting from time t_{10}, the game information should be stored persistently on disk every 10 ticks to avoid data loss in case of a system error. Assume that writing an object onto disk takes two ticks.

The server applies the following changes to the database:

Time	Object	New Value
t_6	O_1	o_1'
t_9	O_2	o_2'
t_{12}	O_3	o_3'
t_{15}	O_1	o_1''
t_{16}	O_3	o_3''
t_{22}	O_2	o_2''
t_{22}	O_3	o_3'''

(a) Outline the procedure of the logging algorithm *Naive Snapshot*.

(b) Outline the procedure of the logging algorithm *Copy-on-Update*.

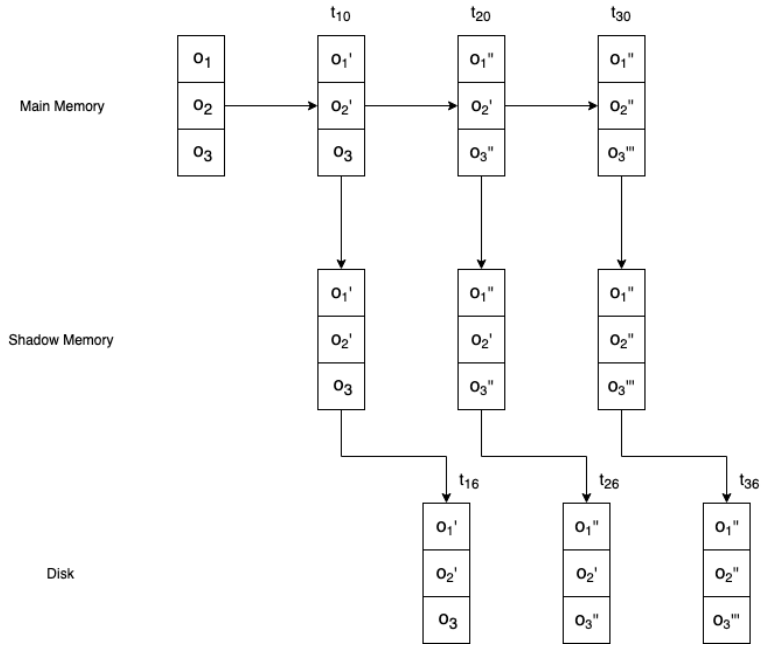
(c) Outline the procedure of the logging algorithm *Wait-Free Zigzag*.

(d) Outline the procedure of the logging algorithm *Wait-Free Ping-Pong*.

(e) Discuss advantages and disadvantages of these methods.

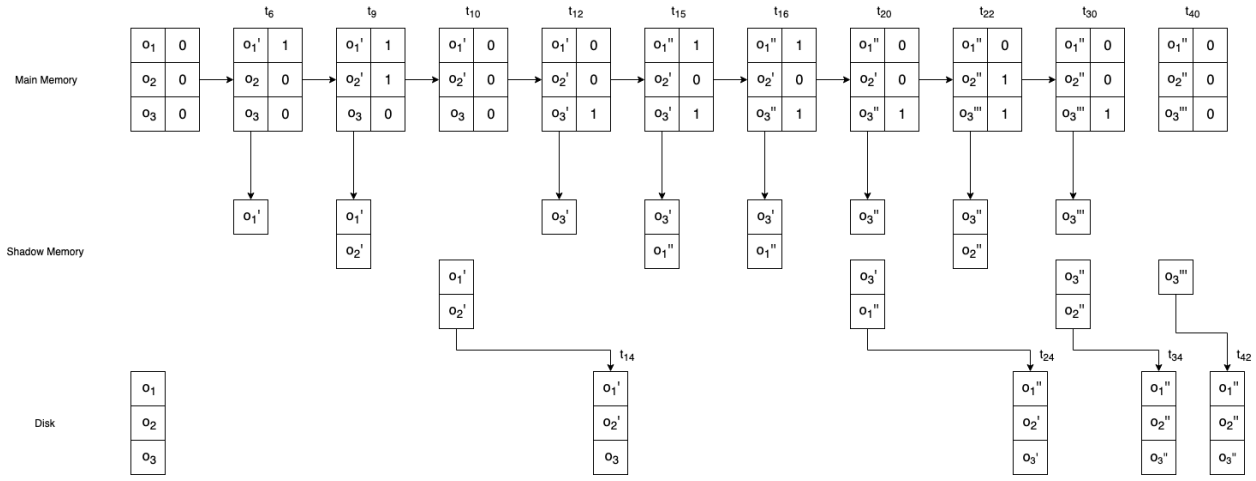
Naive Snapshot

The idea is to copy the entire game state into a shadow memory. From time to time (every tenth tick here) the asynchronous write-thread writes the game state which is available in the shadow memory to disk. Note that this write operation takes 6 ticks in total since we need to write 3 objects (recall: the assumption is that writing a single object requires 2 ticks) to disk. So, this strategy persists the game state batch-wise as updates are accumulated in the shadow memory.



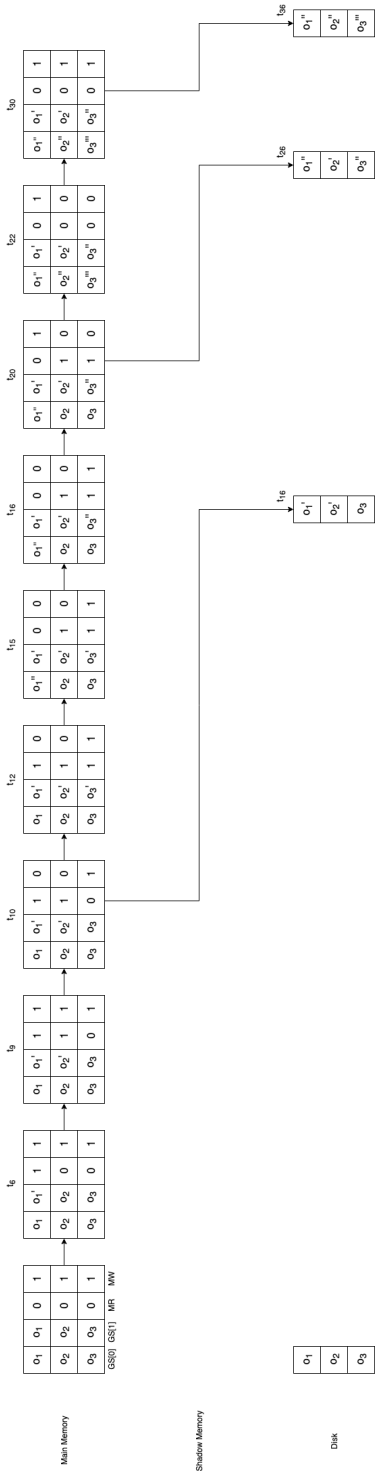
Copy-on-Update

In this approach only updates are copied into the shadow memory. Once an object was updated, it is marked (by using "dirty-bits"). However, the state of an object is only copied into the shadow memory once per period. If an object is updated twice or even more often within a single period, subsequent changes only take effect in the game state (cf. $o_3' \rightarrow o_3''$ at time t_{16}). At checkpoint time, the procedure recognizes subsequent changes on the game state (since the state of an object differs from the state this object has in the shadow memory). Therefore, it first writes all changes tracked in the shadow storage to disk, and resets all markers, resp. dirty-bits. Then, the yet not written object states from the previous period are copied into the shadow memory (e.g., the state o_3'' at time t_{20}).



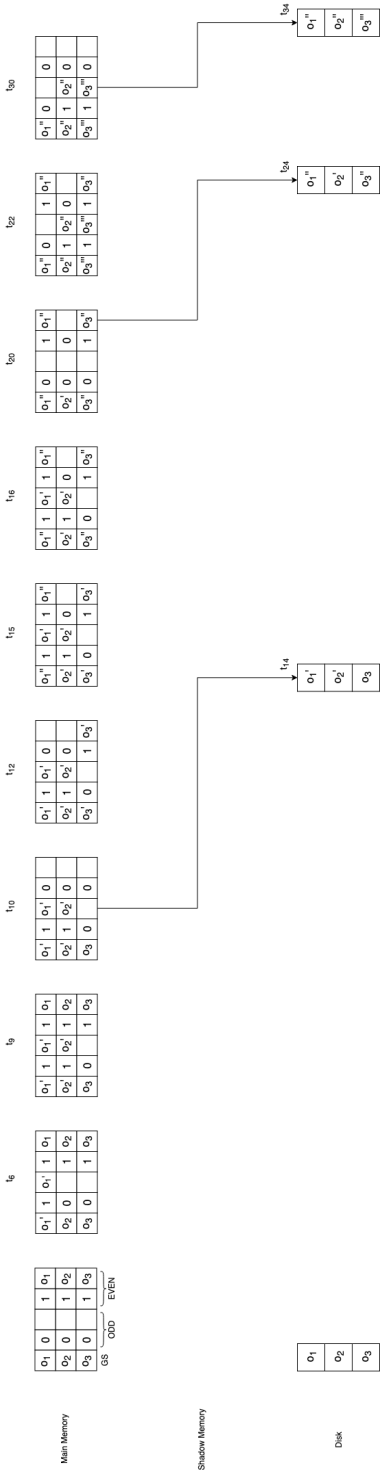
Wait-Free Zigzag

Here, every object contains two flags, i.e., MW (Write-State) and MR (Read-State), each referring to a game state $GS[0]$, resp. $GS[1]$. Generally, the MW-flags remain unchanged during a checkpoint period. If an update to object i appears, the new value is set in $GS[MW_i]$ and the read state flag MR_i is set to MW_i as $GS[MW_i]$ is the most recent state of object i. At the end of a checkpoint period, the MW_i-flags for which $MW_i = MR_i$ holds get flipped. At checkpoint time, the writer-thread then reads the element from $GS[\neg MW_i]$ for object i and writes the objects to disk.



Wait-Free Ping-Pong

For this strategy, one action handling game state GS is used along with two other game states, i.e., persistence-system (read) and persistence-system (write), resp. odd and even. Updates always take place in GS and persistence-system (write). At the end of a checkpoint period persistence-system (write) and persistence-system (read) are swapped. At checkpoint time, the writer-thread then reads from the "new" persistence-system (read) state (as the actual reading process is during the upcoming checkpoint period) and writes the changes which have been tracked there (i.e., the updates from the previous checkpoint period) to disk.



Discussion

- Naive-Snapshot is easiest to implement for very volatile systems with several changes
- The less changes happen, the more advantageous the other methods become
- With Copy-On-Update it might last longer than with the other procedures until a game state is stored on disk (we have seen that the final state in our example is on disk at $t_4 2$, while all other methods have been faster)
- Wait-Free Ping-Pong and Wait-Free Zigzag prevent locking the game entity by the persistence-system
- Wait-Free Ping-Pong also reduces overhead for phase-shifts, but uses a great deal of memory
- Wait-Free ZigZag requires bit comparisons, while Wait-Free Ping-Pong can just flush the bit array for resetting a game state for persistence-system (write)
- On the other hand, Wait-Free Zigzag requires less memory than Wait-Free Ping-Pong (maintaining two game states versus maintaining three game states)
- You may also check slides 15-20 in slide deck 4 (Persistence) for more details