- $n_{ch_{max}} \in (N \cup \{0\})$ — the maximum number of file segments in sc-memory;
- $m_{ch} \in M$ — the object that synchronizes access to CH,$n_{ch_{le}} and n_{ch_{max}}$;
- $tr \in TRM$ — rules (terms) for finding terms in strings;
- TSO — correspondence between string terms and file cell numbers with these sc-memory strings;
- SOF — correspondence between file cell numbers with strings in sc-memory and ostis-system files of which these strings are contents;
- FSO — correspondence between ostis-system files and file cell numbers with sc-memory strings, which are the contents of these strings;
- FSPI = $\{allocate, free, dump, load\}$ — internal programming interface of file storage in scmemory

Unlike sc-element storage, where the cell size is fixed and cells can be allocated in advance as some fixed sequence, this cannot be done in file storage, because cells can store strings of unknown length in advance. Therefore, the accounting of released cells, their fragmentation and defragmentation processes may be more complicated. In this connection, the problem of external fragmentation is not solved in this model, as it is solved in the sc-element storage model.

Each file cell $s_{ij} \in STR, s_{ij} \in ch_i, ch_i \in CH$ has some unique internal address $fa = \langle i, j \rangle \in FA \subset A$. That is, the following statement is always true:

$$\forall s_{ij} \in STR, \exists! fa \in FA : (s_{ij} \in ch_i) \wedge (ch_i \in CH) \wedge$$

$$\wedge (fa = \langle i, j \rangle)).$$

The *allocate* and *free* operations can be defined for file segment cells. Their algorithms are quite simple, so they will not be considered.
File storage specifies operations to enable saving *dump* and loading *load* of all sc memory
This model is focused on the fact that any string can be partitioned into a set of terms, by which, using the T SO mapping, we can determine the indexes of strings that contain these terms [23], [24]. Then, by string indexes it is possible to obtain: from CH — the strings themselves, from SOF — the ostis-system files that contain these lines. Using the FSO mapping it is possible to find the string, which is contained by the given ostis-system file.

$$TSO = TRM \times FA,$$

$$SOF = FA \times FN, FSO = FN \times FA, FN \subset N$$

The model of storage of external information constructions in sc-memory provides:

- storing of the contents of ostis-system files;
- setting the contents to a given ostis file;
- retrieving contents from a specified ostis file;
- retrieving ostis-files by their contents;
- obtaining ostis-system files by their content substring.

The advantages of this model are as follows:

- asymptotic complexity of adding new strings to the storage is $O(1)$ without taking into account the complexity of access time to file storage segments;
- asymptotic complexity of searching ostis-system files and their contents is $O(1)$ without taking intoaccount the complexity of access time to segmentsand cells of the file storage and correspondences between ostis-system files and their contents.

C. Model of storage of subscriptions to events in scmemory

The model of storage of subscription to events in scmemory can be specified by the following tuple

$$RS = \langle V, m_v, RSPI \rangle$$

where

- $V = \{v_1, v_2, ..., v_i, ..., v_n\} i = \overline{1, n}$ is set of subscriptions to events in sc-memory of size n;
- $v_i = \langle e, t_v, a_v, m_v \rangle \in V$ is a subscription to an event in sc-memory;
- e is an sc-element (a cell) in sc-memory that is being "listened";
- $t_v \in T_v$ is a type of event in sc-memory;
- $ag_v \in AG$ is an agent subscribed to an event;
- $m_v \in M$ is an object that synchronizes access to subscription elements;
- RSPI = {*subscribe, unsubscribe, notify*} — internal programming interface of storage of subscriptions to events in sc-memory

All cells in sc-memory can E be listenable $E_l \subseteq E$ and non-listenable $E_{nl} \subseteq E$. For them, the following statements are true:

- $E_l \cup E_{nl} = E, E_n \cap E_{nl} = \varnothing$,
- $E_l \cap E_e = E_l, E_l \cap E_f = \varnothing$,
- $E_{nl} \cap E_e = E_{nl}, E_l \cap E_f = \varnothing$,

$$T_v = \{aoc, aic, roc, ric, re, cc\},$$

where

- *aos* is the event of adding an outgoing sc-connector from the listened sc-element;
- *aic* is the event of adding an incoming sc-connector to the listened sc-element;
- *roc* is the event of removing an outgoing sc-connector from the listened sc-element;
- *ric* is the event of removing an incoming sc-connector from the listened sc-element;
- *re* is the event of removing the listened sc-element;
- *cc* is the event of changing the content of the listened ostis-system file.

For the sets of listened and unlistened cells, the corresponding transitions can be defined in the form of:

- operation of creating a subscription to an event in sc-memory *subscribe* : $E \times T_v \times AG \to E_l$:

$$subscribe(e, t_v, ag_v) = \begin{cases} e_{ij} \in E_l & \text{if } e_{ij} \in E_e \\ Error, & \text{if } e_{ij} \notin E_e \end{cases}$$

- operation of removing a subscription to an event in sc-memory *unsubscribe* : $E_l \to E_{nl}$:

$$unsubscribe(e_{ig}) = \begin{cases} e_{ij} \in E_{nl} & \text{if } e_{ij} \in E_e \\ Error, & \text{if } e_{ij} \notin E_e \end{cases}$$

In addition, the following operation can be defined for the set of listened events$notify$: $(E_l \times C) \cup E_l \to P_w$:

$$notify(e_l, c) = \begin{cases} p \in P_w & \text{if } e_l \in E_l, c \in C, Inc(e_l, c) \\ Error, & otherwise \end{cases}$$

The notify operation can be used to notify (initiate) a process about a new event (creation of an outgoing *arcc* from sc-element e_l, deletion of element e_l, etc.).

The model of storage of subscriptions to events provides:

- storing of event subscriptions in sc-memory;
- ability to subscribe and unsubscribe to an event in sc-memory;
- ability to notify about an event in sc-memory.

D. Model of storage of processes in sc-memory

The model of storage of processes in sc-memory can be defined as

$$PS = \langle P_a, Q_{wp}, n_{map}, PS, PSF, PAG, PSPI \rangle,$$

where

- $P_a \subseteq P$ is the set of active processes in sc-memory;
- Q_{wp} is the queue of processes waiting to start in sc-memory;
- n_{map} is the the maximum possible number of active processes at a given time, $|Q_{wp}| \leq n_{map}$;
- PS is the mapping between active processes and sc-element storage segments;
- PSF is the mapping between active processes and file storage segments;
- PAG is the mapping between active processes and agents;
- RSPI = {*activate, deactivate*} — internal programming interface of storage of sc-memory processes.

$$PS = (P_a \cup P_w) \times S,$$
$$PSF = (P_a \cup P_w) \times CH,$$
$$PAG = (P_a \cup P_w) \times AG.$$

The PS and P SF mappins are used to assign processes to segments of the sc-element storage and file storage. If there are enough free segments in the storage, each process is assigned separate segments in both storages.

All processes in sc-memory P can be waiting $P_w \subseteq P$, active $P_a \subseteq P$ or finished $P_f \subseteq P$.

In this case, each active and waiting process corresponds to an agent that executes it:

$$(\forall p \in P_a, \exists! ag \in AG : (\langle p, ag \rangle \in PAG)),$$

$$(\forall p \in P_w, \exists! ag \in AG : (\langle p, ag \rangle \in PAG)).$$

The following statements are true for all types of processes:

- $P_w \cup P_a \cup P_f = P, P_w \cap P_a \cap P_f = \varnothing$;
- $|P_a| \leq n_{map}$.

Transition between waiting and active processes can be defined as the function $activate : P_w \to P_a$:

$$activate(p_w) =$$

$$= \begin{cases} p_w \in P_a, & \begin{cases} ((\exists s_i \in S_f) \wedge (n_s \leq n_{s_{max}})) \\ ((\exists ch_i \in CH) \wedge (n_{ch} \leq n_{ch_{max}})), \end{cases} \\ Error, & \text{otherwise;} \end{cases}$$

Transition between active and finished processes can be defined as the function $deactivate : P_a \to P_f$:

$$deactivate(p_a) = \begin{cases} p_a \in P_f, & \text{if } p_a \in P_a, \\ Error, & \text{otherwise} \end{cases}$$

A process is considered to be finished if it is not active and not waiting:

$$((p \in P_f) \Leftrightarrow ((\neg p \in P_a) \wedge (\neg p \in P_w))).$$

The model of storage of sc-memory processes provides:

- efficient one-to-one allocation of writer-processes to sc-element storage and file storage segments;
- queuing new processes when the device's processing power is limited, and activating processes from the queue when some active process has finished its work.

E. Model of coordinated access (synchronization) of processes to sc-memory

Each synchronization object $m \in M$ can be represented as [25], [26], [27]:

$$m = \langle c_{ar}, f_{aw}, Q_{rw}, m_u \rangle,$$

where

- c_{ar} is the active reader count;
- f_{aw} is the flag that shows whether a reader is active at a given moment;
- Q_{rw} is the queue of readers and writers;
- m_u is the object used to synchronize access to elements of a given synchronization object.

The queue of readers and writers is a sequence of requests to acquire a particular resource:

$$Q_{rw} = \langle q_1, q_2, ..., q_j, ..., q_m \rangle.$$

Each request q_i includes a unique thread identifier id_j , a thread type (reader or writer) tt_j , and a condition variable that allows messages to be exchanged between processes (threads) cv_j:

$$q_j = \langle id_j, tt_j, cv_j \rangle.$$

This queue ensures that no thread is left hungry.

To coordinate *access* to data structures in sc-memory, mechanisms for acquiring and releasing resources for reader-threads P_r (hereinafter — readers) and writerthreads P_w (hereinafter — writers) are required.

$$P = P_r \cup P_w.$$

These mechanisms should include:

- a reader resource acquisition operation (*acquire_read*) that allows a reader-thread to acquire a synchronization object to start reading the resource and suspend the execution of all other writer-threads while there are readers in the reader-writer queue;
- a reader resource release operation (*release_read*) that allows a reader-thread to release the synchronization object after finishing reading the resource and notify all other writer-threads to execute if there are no active readers in the reader-writer queue after releasing the synchronization object;
- a writer resource acquisition operation (*acquire_write*) that allows a writer-thread to acquire a synchronization object to start modifying the resource and suspend execution of all other reader-threads and writer-threads while there is an active writer in the reader-writer queue;
- a writer resource release operation (*release_write*) that allows a writer-thread to release the synchronization object after the resource modification is complete and notify all other reader-threads and writer-threads to execute if there are no active readers in the reader-writer queue after the synchronization object is released;
- a reader multi-resource acquisition operation (*acquire_read_n*) that allows a reader-thread to acquire multiple synchronization objects for reading in the order necessary to prevent deadlocks;
- a reader multi-resource release operation (*release_read_n*) that allows a reader-thread to release multiple synchronization objects in the reverse order of acquisition;
- a writer multi-resource acquisition operation (*acquire_write_n*) that allows a writer-thread to acquire multiple synchronization objects to modify in the order necessary to prevent deadlocks;
- a writer multi-resource release operation (*release_write_n*) that allows a writer-thread to release multiple synchronization objects in the reverse order of acquisition.

Allocation of sc-memory to writers can be done segment-by-segment using a specialized table T_{ps}, which allows to determine whether a given vacant sc-memory segment has been acquired by another writer:

$$s : S \to S_v, T_{ps} \subseteq P_w \times S_v.$$

Let us recall that a free sc-memory segment can be either a segment with vacant cells or a segment with released cells. When allocating sc-memory, the first thing that is done is to search for vacant segments that are not used by other writers. If no such segments are found, new segments are allocated. If there is no available space in the sc-memory for new segments, writers can use segments from the list of engaged vacant segments.

To ensure coordinated read access to segments, each segment contains a unique synchronization object.

$$m_s : S \times M \to S_m,$$

$$m_{ch} : CH \times M \to CH_m.$$

In addition to segments, synchronization objects are also temporarily assigned to sc-memory cells and events to be registered in it. Synchronization objects of sc-memory cells can be stored in a specialized table T_{em}. These objects are used to synchronize access to the sc-element information contained in the sc-memory element:

$$T_{em} \subseteq A \times M.$$

These objects synchronize the subscription and unsubscription to events through a single table, as well as the initiation of the sc-agents themselves

$$v_i = \langle t_v^i, A_v^i, m_v^i \rangle, t_v^i \in T_v, A_v^i \subseteq A_v, m_v^i \in M,$$

$$v_m : V \times M \to V_m.$$

The model of synchronization of process access to sc-memory provides:

- parallel access to sc-memory, i.e. the possibility ofparallel execution of actions in sc-memory withoutviolating correctness of data structures in it;