сообщения
объединенного
института
ядерных
исследований
дубна

E11-83-588

M.Rudalics

DESIGN AND IMPLEMENTATION OF A WORKSTATION DEPENDENT SEGMENT STORAGE MANAGER

1983

1. INTRODUCTION AND PREVIOUS WORK*

The workstation dependent segment storage (WDSS) concept is explained in (GKS 82). The workstation dependent segment storage manager (WDSSM) coordinates the allocation of freshly created segments with the recycling of deleted ones in the memory of an intelligent graphics terminal (IGT) (Lei 83).

In addition to the proper WDSS the WDSSM has to manage the display list which is composed of representations, i.e. those items which are generated for putting segments on the screen. A concise description of all objects handled by the WDSSM may be found in (Rud 83).

Special attention in the design of the WDSSM has been given to the management of the display list, as this (or parts of it) is thrown away with every regeneration of the display image. Regenerations may occur selectively, (i.e. for one segment at a time, e.g. when applying a new segment transformation), or globally (i.e. for all segments stored at the WDSS, e.g. when updating the current workstation transformation). Beforemost the latter, i.e. global regenerations, rely on the performance of the WDSSM, as the entire WDSS has to be traversed, old representations have to be deleted (preferable one at a time, for the reason not to traverse the WDSS twice), and new representations have to be created. All these actions have to be performed under the condition, that the display process continuously passes over the display list and is not disturbed by regeneration.

For these reasons it has been decided to build a real-time (incremental) garbage collection system, which is supposed to keep the time required for allocating a new representation sufficiently low.

Garbage collection has been introduced in list processing languages by McCarthy (McC 60). The principles of garbage collection are, see (Coh 81), to identify reclaimable storage space (this is generally referred to as marking), and to make this space available to the user (this is generally referred to as collecting).

---

* A short version of this paper will be presented at the 5th International Conference on Mathematical Modelling, Programming and Mathematical Methods for Solving Physical Programs. Dubna 1983.

Due to the fact that in classic list processing systems computation has to be suspended for the time required by garbage collection, real-time garbage collection has been proposed as an alternative (Knuth credits this idea to Minsky, (Knu 73) pp. 422, 594). The algorithms developed for parallel garbage collection, by Steele (Ste 75), Dijkstra et.al. (Dij 76), Kung et.al. (Kun 77), and analyzed by Wadler (Wad 76), have in common, that they operate only on homogeneous memory spaces,i.e. all objects managed by the garbage collector have to be of the same size.

The general case of handling variable sized objects in real-time is treated within the concept of copying garbage collection (Bak 78),(Lie 83).Copying garbage collection has the disadvantage that it requires two times the storage used by a non-copying garbage collector. Although this drawback may not be considered to serious when one thinks of the advantages of including compact coding techniques within a copying garbage collector implementation (see (Bak 78) for a listing of all those concepts, like CDR-coding, etc.), and may be estimated a minor one in the environment of large virtual memory systems (Bis 77), it will not do for our implementation, considering the IGT's small core capacity (48K bytes) and the absence of secondary storage.

The compactification attempted by some of the systems mentioned above, in an extra phase by Steele (Ste 75), or as a property of allocation by Baker (Bak 78), is not suited for our purposes, as it either requires items to be homogeneous (Ste 75), or relies on the copying concept (Bak 78).

In the WDSSM compactification is also inhibited by the fact, that the microprogrammed display process is not (and should not be) capable of performing semaphore-type operations which are essential for excluding this process from accessing those items on the display list which are currently relocated and therefore temporarily unsave.

We will not, however, exclude a future version of a WDSSM which will rely on a twofold strategy, where the proper WDSS (residing in one partition of the memory) is handled by a garbage collector with an additional compactification phase, while the display list (residing in another, physically distinct partition of the memory) is managed by something close to the present version. Unresolved remains the question where to allocate bundles (Rud 83), when employing this strategy: in the display list partition, splitted between WDSS and display list partition, or in a third partition?

2. DESIGN OF THE WDSSM

A schematic description of the interfaces between host and workstation resident routines implementing GKS functions, the WDSSM, and the display process is given in Figure 1. Declarations and algorithms are described in PL-M/80 (our implementation language) slightly modified for presentation.

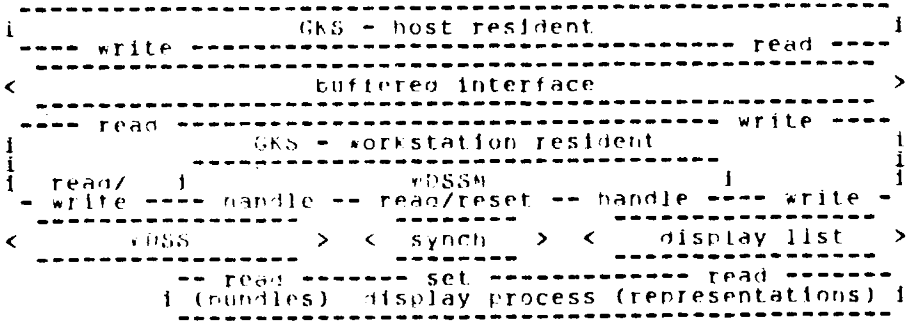

Fig.1. Interfaces between GKS routines, WDSSM and display process.

The algorithms of the WDSSM recognize four different groups of items, respectively:

(1) User Items

DCL user_item STRUCT (type BYTE, size INT);

User items are headed by a type field and a size field. For some user items of fixed size (known to the procedure get_size) size may be an implicit function of type. Types for user items may run from four to 255 (a continuous subrange is supposed to be used in an implementation).

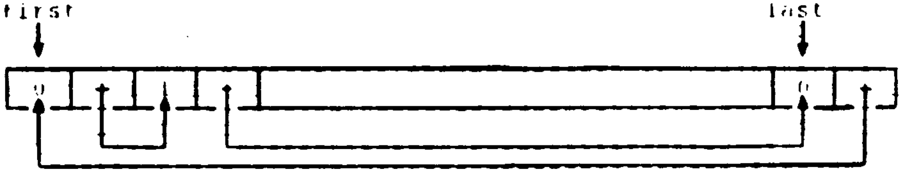

Fig.2. Initial memory layout.

(2) Free List Items

DCL free_list_item STRUCT (size INT, next PTR);

Free list items are circularily linked via their next field and do not contain a type field. Algorithms are supposed to recognize free list items by their relative position on the free list. The free list is initialized as indicated in Figure 2. First and last are two distinguished free list items located at the low and high end of the memory respectively. As their size always remains equal zero the allocation algorithm (cons) ignores them, consequently the free list virtually is never empty. The size of the free list item following first initially comprises the whole memory.

(3) Loose Items

DCL loose_item BYTE;

Loose items are free items shorter than four bytes and consist of the size of the item only, appropriately their type runs from one to three. Loose items are not on the free list and may therefore not be used for allocation, however, they may serve for recombining free list items.

(4) Marked Items

DCL mark_item STRUCT (type BYTE, size INT);

Marked items are generated by the marking algorithm, they suitably have type zero.

The WDSSM operates on three lists: The list of items to be deleted (del_list), the list of items to be marked (mark_list), and the list of free items (free_list). Del_list is built by routines which implement GKS functions (delete segment, redraw all segments on workstation, clear workstation, etc.). Segments and representations enter this list in LIFO order. Mark_list is operated upon by the mark procedure. After each collection cycle mark_list and del_list change (in one, indivisible operation) their relative roles, the past del_list becomes the new mark_list and del_list is initialized to the empty list. Marking will only be performed when at least one item had been deleted, i.e. entered the previous del_list.

The mark procedure does not mark items in use, but discarded items which are accessible from a deleted segment or representation. In the WDSSM explicit deletion does not require the overhead usually associated with it, as only the headers of segments and representations have to be deleted (i.e. added to del_list) explicitely. Marking is performed nonrecursively, see (Tho 72); mark_list serves as an auxiliary list for tracing out inserted segments. Marking of more specific objects (instances, s_functions, and r_functions) is not described in detail. Note however, that for example mark_ins contains a critical section where it has to be synchronized in an appropriate manner with the insert procedure.

A special construct in the WDSSM is the synchronization flag (synch) which is reset by the WDSSM and is set by the display process after completion of each refresh cycle (at least all 20 msecs). Synch (when false) keeps the mark procedure from marking items the display process may still be proceeding. Performing synchronization in this clumsy way is necessitated by the fact that the (microprogrammed) display process may not perform semaphore-type operations, but is only capable of a global assertion like: "at this moment I am outside the display list".

The collector (coll) has two free list items (termed prev and next) with size zero (so the cons procedure does not consider them for allocation) wander from first to last. Whenever it encounters a free list item, coll sets (in one, indivisible operation) the item's size to zero. When a new free list item is created, temporarily a third free list item with size zero (termed free) may exist, as outlined in Figure 3.

Cons uses for allocation the next-fit (also modified first-fit, or FF/Rover) method, which is described in (Knu 73). Next-fit uses memory worse than first-fit and best-fit as reported by (Bay 77), but has significantly better performance characteristics as simulation results document (Nie 77). In the real-time system next-fit requires two more indivisible operations in the collector, when during recombination a free list item is swallowed and the rotating starting point for searching (cons_base) has to be relocated, this process is described in Figure 4.

Cons ping-pongs the search of a host item (i.e. an item on the free list capable to accomodate an object of a requested type and size) between two structures (a_free and b_free). This eliminates the need for saving the address of the previous (respectively to the host) free list item in order to update its next field when the host item has been exhausted by allocation.

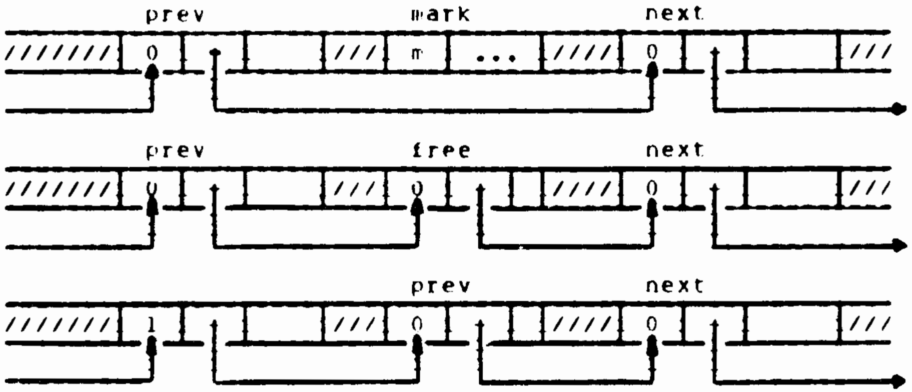

Fig.3. A new free list item is created (slashes indicate items in use).

The problem of remembering the previous item does not occur when the free list is organized in a double linked manner. This would, however, introduce additional overhead and enlarge the size of the smallest free list item to six bytes.

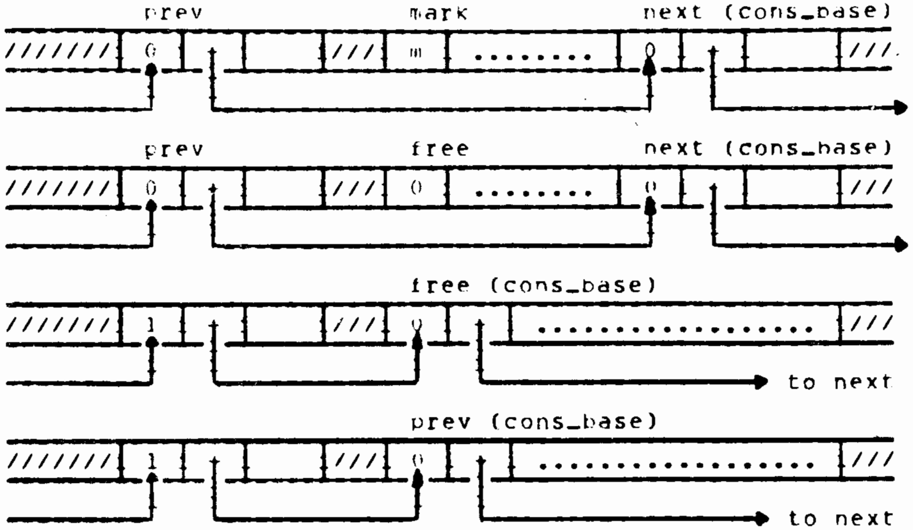

Fig.4. A free list item is swallowed and cons base has to be relocated (slashes indicate items in use).

3. IMPLEMENTATION OF THE WDSSM

Two methods for implementing real-time garbage collectors have been proposed:

The serial method has list processing and garbage collection run on one and the same processor. In our implementation this means that the WDSSM acts as a subordinate to the regeneration process, invoked when time consuming operations (e.g. transformations, which have to be performed by the arithmetics processor) take place. Collection is assumed to be performed iteratively, i.e. coll will advance by one or two free list items in one step.

The parallel method has list processing run on one processor and garbage collection on another. Collection is supposed to run uninterrupted but for synchronization operations. The parallel case is not implementable without modifications of our present cons procedure, as a_free and b_free (and their relative positions on the free list) may be altered by the coll procedure in between two searches. At least the two pointer scheme of cons is not feasible in the parallel method, a double linked implementation of the free list seems to be more appropriate for this purpose. However the relative merits of the parallel method seem doubtfull, as the coll procedure is likely to be locked out anyway during the whole duration of a cons operation.

Our present implementation adopts a quasi-parallel solution which has regeneration run on one processor, cons and garbage collection on another. Garbage collection is interrupted only when regeneration needs to allocate a new object in the memory. In this case cons is called and may serve the request. When cons runs out of space, regeneration terminates gracefully, i.e. the display image will remain incomplete but the proper WDSS will remain untouched. In this situation the user may free some space (for example by changing the current workstation transformation or by displaying some segments in store mode) and resume execution.

When performing indivisible operations the WDSSM need not disable its interrupts to lock out requests which must not interfere with the collector, but use a special test and set logic to lock out other processes only when critical sections are concerned. In this context it should be remarked, that operations on data of type INT or PTR do not require any interlocking on the IGT, as they have been rendered indivisible by altering the hardware mechanisms for accessing the common memory.

4. CONCLUSION

A workstation dependent segment storage management system suited for implementation on a multiprocessor based graphics workstation has been exhibited. The WDSSM presently operates in the testbed of a simulator implemented on an intellec mds development system. Work proceeds to incorporate the WDSSM in the software of an intelligent graphics terminal, which is currently under construction at the JINR.

Given certain restrictions, the algorithms constituting the WDSSM may be recommended for use in general purpose memory management systems like the one proposed in (Car 79). Optimal performance, however, will be achieved only when the objects to be handled by the WDSSM may accomodate at least a size and a pointer field.

```
                            % Global declarations: %
DCL mark_type LIT '0';      % The type of a marked item. %
DCL min_size LIT '4';       % Minimum for free list item. %
DCL first_base PTR;         % Low end of memory. %
DCL last_base PTR;          % High end of memory. %
DCL cons_base PTR;          % Cons starts from here. %
DCL synch BYTE;             % Set by display process. %
DCL del_base PTR;           % Root of del_list. %
                            % The procedure get_size %
get_size: PROCEDURE (item_base) INT;
 DCL item_base INT;         %  returns the size of the %
END get_size;               %  item located at item_base. %
                            % Ex_loop synchronizes the %
ex_loop: PROCEDURE;         %  WDSSM with the display %
                            %  and deletion processes. %
DCL mark_base PTR;          % Root of mark_list. %
DCL first BASED first_base STRUCT
 (size INT, next PTR);      % First free list item. %
DCL free BASED free_base STRUCT
 (size INT, next PTR);      % Initially the whole memory. %
DCL last BASED last_base STRUCT
 (size INT, next PTR);      % Last free list item. %
                            % [Initialization at reset] %
free_base = first_base + min_size;
first.size = 0;
first.next = free_base;
free.size = last_base - free_base;
free.next = last_base;
last.size = 0;
last.next = first_base;
cons_base = free_base;      % Start cons at free_base. %
del_base = NIL;             % Set del_list empty. %
mark_base = NIL;            % Set mark_list empty. *
synch = FALSE;              % Reset synch. %
                            % [End of initialization] %
DO FOREVER;
 IF mark_base <> NIL
  THEN DO;                  % Mark_list not empty. %
   DO WHILE (synch = FALSE);
   END;                     % Wait for synch. %
   CALL mark (mark_base);   % Mark items on mark_list. %
                            % [Indivisible begin] %
   mark_base = del_base;    % Del_list becomes mark_list. %
   del_base = NIL;          % Set del_list empty. %
                            % [Indivisible end] %
   synch = FALSE;           % Reset synch. %
   CALL coll;               % Collect. %
  END;
 ELSE DO;                   % Mark_list empty. %
                            % [Indivisible begin] %
  mark_base = del_base;     % Del_list becomes mark_list. %
  del_base = NIL;           % Set del_list empty. %
                            % [Indivisible end] %
  synch = FALSE;            % Reset synch. %
 END;
END;
END ex_loop;
```

*

```
                             % Mark manipulates a list %
mark: PROCEDURE (item_base);%  of deleted items, which %
                             %  have a type and a prev %
DCL item_base PTR;           %  field, where latter points %
DCL del BASED item_base STRUCT
 (type BYTE, prev PTR);      %  to the previously deleted %
decl mark BASED item_base STRUCT
 (type BYTE, size INT);      %  item. Mark marks all items %
DCL prev_base INT;           %  on this list and some %
                             %  items accessible from it. %
DO WHILE (item_base <> NIL);% while mark_list is not %
 IF del.type = seq_type      %  empty, do: %
  THEN DO;                   % For every segment %
   CALL mark_ins;            %  mark imported segments %
   CALL mark_s_fun;          %  and mark s_funs. %
  END;
 ELSE IF del.type = rep_type
  THEN                       % For every representation %
   CALL mark_r_fun;          %  mark r_funs. %
 prev_base = del.prev;       % Remember prev item on list, %
 mark.size = get_size (item_base);
 mark.type = mark_type;      %  store size in mark.size, %
 item_base = prev_base;      %  mark item in mark.type, %
END;                         %  and proceed previous item. %
END mark;
```

*

```
                             % Coll may occupy two (or %
coll: PROCEDURE;             %  three) consecutive items %
                             %  on the free list: Prev, %
DCL prev_base PTR;           %  already proceeded, with %
DCL prev BASED prev_base STRUCT
 (size INT, next PTR);       %  prev.size zero, its actual %
DCL prev_size INT;           %  size remembers prev_size; %
DCL free_base PTR;           %  free, just created, with %
DCL free BASED free_base STRUCT
 (size INT, next PTR);       %  free.size zero, its actual %
DCL free_size INT;           %  size remembers free_size; %
DCL next_base PTR;           %  next, yet to be proceeded, %
DCL next BASED next_base STRUCT
 (size INT, next PTR);       %  next.size zero, its actual %
DCL next_size INT;           %  size remembers next_size. %
DCL item_base PTR;           % The current item. %
DCL item_type BASED item_base BYTE;
DCL item_size BASED item_base INT;
DCL mark BASED item_base STRUCT
 (type BYTE, size INT);      % An item produced by mark. %
DCL loose BASED free_base BYTE;
DCL add BYTE;                % A switch ... %
                             % To make next inaccesible %
acquire_next: PROCEDURE;     % [Indivisible begin] %
 next_base = next.next;      %  for cons: Locate it, %
 next_size = next.size;      %  remember its size first, %
 next.size = 0;              %  and set it zero after. %
END acquire_next;            % [Indivisible end] %
                             % To insert an new item into %
replace_prev: PROCEDURE;     %  the free list: %
 free.size = 0;              %  Set its size to zero, %
 free.next = next_base;      %  have it point to next, and %
 prev.next = free_base;      %  have prev point to it. Now %
 prev.size = prev_size;      %  restore prev's size, and %
END replace_prev;            %  cons may use prev. %
                             % To release prev: Restore %
release_prev: PROCEDURE;     %  its actual size (which %
 prev.size = prev_size;      %  has been remembered by %
END release_prev;            %  prev_size). Cons may now %
                             %  use prev. %
prev_base = first_base;      % Base prev at first. %
prev_size = 0;               %  (size of first is zero), %
next_base = prev_base;       %  let next be the next free %
CALL acquire_next;           %  list item after first. %
item_base = first_base + min_size;%
```

```
DO forever;                   % Loop with new current item. %
 IF item_base = next_base
  THEN DO;                    % Hit next ... %
   CALL release_prev;
   IF item_base = last_base
    THEN                      % Next is already last - %
     RETURN;                  %  done. %
   ELSE DO;                   % Next is not last - %
    prev_base = next_base;    %  next becomes prev, %
    prev_size = next_size;
    CALL acquire_next;        %  next.next becomes next. %
    item_base = prev_base + prev_size;
    DO forever;               % Loop with new current item. %
     IF item_base = next_base
      THEN DO;                % Hit next ... %
       IF item_base = last_base
        THEN DO;              % Next is already last - %
         CALL release_prev;
         RETURN;              %  done. %
        END;
       ELSE DO;               % Next is not last - swallow. %
        prev_size = prev_size + next_size;
                              % [Indivisible begin] %
        IF item_base = cons_base
         THEN                 % Hit cons_base -> relocate. %
          cons_base = prev_base;
                              % [Indivisible end] %
        CALL acquire_next;    % We need a new next. %
        prev.next = next_base;
       END;
      END;
     ELSE IF item_type = mark_type
      THEN                    % Current item is marked. %
       prev_size = prev_size + mark.size;
     ELSE IF item_type < min_size
      THEN                    % Current item is loose. %
       prev_size = prev_size + item_type;
     ELSE DO;                 % Current item still in use, %
      item_base = item_base + get_size (item_base);
      LEAVE_LOOP;             %  -> no further appending. %
     END;
     item_base = prev_base + prev_size;
    END;
   END;
  END;
 ELSE IF item_type < min_size
  THEN DO;                    % Current item is loose or %
   free_base = item_base;     %  marked, may be combined to %
   IF item_type = mark_type   %  a new free list item. %
    THEN DO;                  % Current item is marked, %
     free_size = mark.size;
     IF free_size < min_size
      THEN                    % Marked item yet too short %
       add = FALSE;           %  to become free list item. %
     ELSE DO;                 % Marked item long enough to %
      CALL replace_prev;      %  become free list item, %
      add = TRUE;             %  -> free replaces prev. %
     END;
    END;
   ELSE DO;                   % A loose item is not added %
    free_size = item_type;    %  to the free list, but its %
    add = FALSE;              %  size is remembered. %
   END;
   item_base = free_base + free_size;
```

```
   DO FOREVER;
    IF item_base = next_base
     THEN DO;                 % Hit next ... %
      IF item_base = last_base
       THEN DO;               % Next is already last. %
        IF add = FALSE
         THEN DO;             % Item before last is loose. %
          CALL release_prev;
          loose_size = free_size;
         END;
        ELSE                  % Item before last is free. %
         free.size = free_size;
        RETURN;               % Done. %
       END;
      ELSE DO;                % Update free_size. %
       free_size = free_size + next_size;
       CALL acquire_next;     % Get next free item. %
       IF add = FALSE
        THEN DO;              % Point to next from prev. %
         prev.next = next_base;
                              % [Indivisible begin] %
         IF item_base = cons_base
          THEN                % Hit cons_base -> relocate. %
           cons_base = prev_base;
        END;                  % [Indivisible END ] %
       ELSE DO;               % Point to next from free. %
        free.next = next_base;
                              % [Indivisible begin] %
        IF item_base = cons_base
         THEN                 % Hit cons_base -> relocate. %
          cons_base = free_base;
       END;                   % [Indivisible END ] %
      END;
     END;
    ELSE IF item_type = mark_type
     THEN                     % Current item is marked. %
      free_size = free_size + mark.size;
    ELSE IF item_type < min_size
     THEN                     % Current item is loose. %
      free_size = free_size + item_type;
    ELSE DO;                  % Current item still in use %
     item_base = item_base + size (item_base);
     LEAVE_LOOP;              %  -> no further appending. %
    END;
    IF (free_size >= min_size) AND (add = FALSE)
     THEN DO;                 % Combined item long enough %
      CALL replace_prev;      %  to become free list item, %
      add = TRUE;             %  -> free replaces prev. %
     END;
    item_base = free_base + free_size;
   END;
   IF add = FALSE
    THEN                      % Item too short -> loose. %
     loose_size = free_size;
   ELSE DO;                   % Item long enough, free now %
    prev_base = free_base;    %  nominally becomes prev. %
    prev_size = free_size;
   END;
  END;
 ELSE                         % Current item still in use. %
  item_base = item_base + size (item_base);
END;

END coll;
```

*

```
                            % Cons tries to allocate %
cons: PROCEDURE (req_type, req_size) PTR;
DCL req_type BYTE;          %  an item of type req_type %
DCL req_size INT;           %  and size req_size, %
DCL req_base PTR;           %  initialize it, and return %
DCL req BASED req_base STRUCT
 (type BYTE, size INT);     %  a pointer to it. %
DCL a_free_base PTR;        % The host item will be %
DCL a_free BASED a_free_base STRUCT
 (size INT, next PTR);      %  either a_free or b_free, %
DCL b_free_base PTR;
DCL b_free BASED b_free_base STRUCT
 (size INT, next PTR);      %  after allocation its %
DCL rem_size INT;           %  size will be rem_size. %
                            % Init_req initializes the %
init_req: PROCEDURE (prev_base, host_base) PTR;
                            %  allocated item. %
DCL host_base PTR;          % The host item. %
DCL host BASED host_base STRUCT
 (size INT, next PTR);
DCL prev_base PTR;          % The free item preceding %
DCL prev BASED prev_base STRUCT
 (size INT, next PTR);      % The host item. %
                            % Satisfy a request: %
rem_size = host.size - req_size;
IF rem_size >= min_size
 THEN DO;                   % The host item remains on %
  host.size = rem_size;     %  the free list, with size %
  cons_base = host_base;    %  reduced to rem_size. %
 END;
ELSE DO;                    % The host item has been %
 host.size = rem_size;      %  exhausted, a loose item or %
 cons_base = host.next;     %  nothing remains, and prev %
 prev.next = cons_base;     %  gets a new next. %
END;
req_base = host_base + rem_size;
req.type = req_type;        % The allocated item gets a %
req.size = req_size;        %  type and a size, its %
RETURN (req_base);          %  address is returned. %
END init_req;

b_free_base = cons_base;    % Base b_free on cons_base, %
a_free_base = b_free.next;  % A_free is b_free's next. %
DO FOREVER;                 % Consider a_free, %
 IF a_free.size >= req_size
  THEN                      % ... fits. %
   RETURN (init_req (b_free_base, a_free_base));
 ELSE DO;                   % ... does not fit. %
  IF a_free_base = cons_base
   THEN                     % Full cycle round, %
    RETURN (NIL);           % Cannot satisfy request. %
  ELSE                      % B_free is a_free's next. %
   b_free_base = a_free.next;
 END;                       % Consider b_free, %
 IF b_free.size >= req_size
  THEN                      % ... fits. %
   RETURN (init_req (a_free_base, b_free_base));
 ELSE DO;                   % ... does not fit. %
  IF b_free_base = cons_base
   THEN                     % Full cycle round, %
    RETURN (NIL);           % Cannot satisfy request. %
  ELSE                      % A_free is b_free's next. %
   a_free_base = b_free.next;
 END;
END;

END cons;
```

*

REFERENCES

(Bak 78) Baker, H.G.jr. List Processing in Real Time on a Serial Computer. Comm.ACM 21, 4 (April 1978), 280-294.

(Bay 77) Bays, C. A Comparison of Next-fit, First-fit, and Best-fit. Comm. ACM 20, 3 (March 1977), 191-192.

(Bis 77) Bishop, P.B. Computer Systems with a Very Large Address Space and Garbage Collection. Ph.D. Thesis, MIT/LCS/TR-178, Cambridge, Mass., May 1977.

(Car 79) Cardelli, L. A General Purpose Memory Management System. Unpublished Notes, 1979.

(Coh 81) Cohen, J. Garbage Collection of Linked Data Structures. ACM Computing Surveys 13, 3 (Sept. 1981), 341-367.

(Dij 76) Dijkstra, E.W., Lamport,L., Martin,A.J., Scholten, C.S., and Steffens,E.F.M. On-the-fly Garbage Collection: An Exercise in Cooperation. In: Language Hierarchies and Interfaces. F.L. Bauer and K.Samalson (eds.), Springer-Verlag, New York, 1976, pp. 43-56. Also appeared in: Comm.ACM 21, 11 (Nov. 1978), 966-975.

(GKS 82) Draft International Standard ISO/DIS 7942, Information Processing Graphical Kernel System (GKS), Functional Description, Version 7.2, NI-5 9/1-83, Nov. 1982

(Knu 73) Knuth,D.E. The Art of Computer Programming, Vol. I: Fundamental Algorithms. Addison Wesley, Reading, Mass., 1973.

(Kun 77) Kung,H.T., Song,S.W. An Efficient Parallel Garbage Collection System and Its Correctness Proof. Department of Computer Science Report, Carnegie-Mellon Univ., Pa., Sept. 1977.

(Lei 83) Leich,H., Levchanovsky,V., Prikhodko,V. Multimikroprozessorsystem zur Steuerung eines Intelligenten Graphischen Terminals (IGT). To appear in: Nachrichtentechnik Elektronik. Also appeared as: JINR, P10-83-7 (in Russian), Dubna, 1983.

(Lie 83) Lieberman,H., Hewitt,C. A Real-Time Garbage Collector Based on the Lifetimes of Objects. Comm. ACM 26, 6 (June 1983), 419-429.

(McC 60) McCarthy,J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, I. Comm.ACM 3, 4 (April 1960), 184-195.

(Nie 77) Nielsen,N.R. Dynamic Memory Allocation in Computer Simulation. Comm.ACM 20, 11 (Nov. 1977), 864-873.

(Rud 83) Rudalics,M. An Intelligent Graphics Terminal's Intermediate Database. In: Proc. of Eurographics'83, North-Holland Pub., Amsterdam, 1983, pp. 383-392.

(Ste 75) Steele,G.L. jr. Multiprocessing Compactifying Garbage Collection. Comm.ACM 18, 9 (Sept. 1975), 495-508. Corrigendum: Comm.ACM 19, 6 (June 1976), 354.

(Tho 72) Thorelli,L.E. Marking Algorithms. BIT, Nordisk Tidskrift for Informationsbehandling 12, 4 (1972), 555-568.

(Wad 76) Wadler,P.L. Analysis of an Algorithm for Real Time Garbage Collection. Comm.ACM 19, 9 (Sept. 1976), 491-500.

Received by Publishing Department
on August 12, 1983.

WILL YOU FILL BLANK SPACES IN YOUR LIBRARY?

You can receive by post the books listed below. Prices - in US $, including the packing and registered postage

D-12965	The Proceedings of the International School on the Problems of Charged Particle Accelerators for Young Scientists. Minsk, 1979.	8.00
D11-80-13	The Proceedings of the International Conference on Systems and Techniques of Analytical Computing and Their Applications in Theoretical Physics. Dubna, 1979.	8.00
D4-80-271	The Proceedings of the International Symposium on Few Particle Problems in Nuclear Physics. Dubna, 1979.	8.50
D4-80-385	The Proceedings of the International School on Nuclear Structure. Alushta, 1980.	10.00
	Proceedings of the VII All-Union Conference on Charged Particle Accelerators. Dubna, 1980. 2 volumes.	25.00
D4-80-572	N.N.Kolesnikov et al. "The Energies and Half-Lives for the α - and β-Decays of Transfermium Elements"	10.00
D2-81-543	Proceedings of the VI International Conference on the Problems of Quantum Field Theory. Alushta, 1981	9.50
D10,11-81-622	Proceedings of the International Meeting on Problems of Mathematical Simulation in Nuclear Physics Researches. Dubna, 1980	9.00
D1,2-81-728	Proceedings of the VI International Seminar on High Energy Physics Problems. Dubna, 1981.	9.50
D17-81-758	Proceedings of the II International Symposium on Selected Problems in Statistical Mechanics. Dubna, 1981.	15.50
D1,2-82-27	Proceedings of the International Symposium on Polarization Phenomena in High Energy Physics. Dubna, 1981.	9.00
D2-82-568	Proceedings of the Meeting on Investigations in the Field of Relativistic Nuclear Physics. Dubna, 1982	7.50
D9-82-664	Proceedings of the Symposium on the Problems of Collective Methods of Acceleration. Dubna, 1982	9.20
D3,4-82-704	Proceedings of the IV International School on Neutron Physics. Dubna, 1982	12.00

Orders for the above-mentioned books can be sent at the address:
Publishing Department, JINR
Head Post Office, P.O.Box 79 101000 Moscow, USSR

SUBJECT CATEGORIES OF THE JINR PUBLICATIONS

Index	Subject
1.	High energy experimental physics
2.	High energy theoretical physics
3.	Low energy experimental physics
4.	Low energy theoretical physics
5.	Mathematics
6.	Nuclear spectroscopy and radiochemistry
7.	Heavy ion physics
8.	Cryogenics
9.	Accelerators
10.	Automatization of data processing
11.	Computing mathematics and technique
12.	Chemistry
13.	Experimental techniques and methods
14.	Solid state physics. Liquids
15.	Experimental physics of nuclear reactions at low energies
16.	Health physics. Shieldings
17.	Theory of condenced matter
18.	Applied researches
19.	Biophysics

Рудалич М. E11-83-588

Разработка и реализация системы управления памяти приборно-зависимой базы данных для сегментов

Представлена система управления приборно-зависимой базы данных для сегментов, которая распределяет и вновь использует изменяемые части памяти в реальном масштабе времени. Система обеспечивает управление приборно-зависимой базой данных для сегментов, как описано в стандарте ГКС, и списком дисплейных команд для векторного дисплея в памяти интеллектуального графического терминала, реализованного на основе мультипроцессорной системы. Детально описаны алгоритмы, особое внимание уделено механизмам синхронизации и связи между процессами.

Работа выполнена в Лаборатории вычислительной техники и автоматизации ОИЯИ.

Сообщение Объединенного института ядерных исследований. Дубна 1983

Rudalics M. E11-83-588

Design and Implementation of a Workstation Dependent Segment Storage Manager

A workstation dependent segment storage management system which allocates and recycles variable portions of memory under real-time conditions is presented. The system is capable of handling a workstation dependent segment storage, as described in the Graphical Kernel System, and the display list for a vector display, in the memory of a multiprocessor based implementation of an intelligent graphics terminal. Algorithms are presented in detail with particular attention to mechanisms for interlocking and communication between processes.

The investigation has been performed at the Laboratory of Computing Techniques and Automation, JINR.

Communication of the Joint Institute for Nuclear Research. Dubna 1983