DPMI Persistent Object Store (DPOS)

Paul Cockshott

October 7, 1999

Orthogonal persistence[Atkinson83][Cockshott83] is a technique by which a program sees a uniform view of data irrespective of its lifetime. Thus any datastructure that a programmer can declare in a programming language can be made to persist over several progam invocations without the programmer having to make any special effort to ensure this. Reviews of implementation techniques for persistent stores are provided in [Cockshott85] and [Suzuki94]. Its attraction from the standpoint of database implementation is that, by concentrating the data being worked on in fast random access store it offers the promise of considerable performance gains when compared to orthodox designs which are centered on disk files. If one assumes that ones data is basically stored in RAM not disk, the cost of following a pointer becomes' modest. A consequence is that it becomes possible to use more sophisticated and complex datastructures without having to pay an excessive performance penalty. In particular, the use of data compression becomes a more attractive option. This is both because a compressed database will have a smaller working set and thus perform better with a given RAM size, and also because the costs of indirection, which are implied by the use of dictionaries to tokenise the database, now become supportable.

1 Difficulties in the use of persistence

In a properly developed persistent programming language the burden of arranging for transfers of data between its long term storage and the volatile memory of the computer is fully automated. Data-structures declared in a program, automatically persist between program invocations and may be exchanged between different programs, without the application programmer having to write any instructions to store data onto non-volatile media. In addition, a garbage collection facility will automatically recover memory locations occupied by data that is now unwanted.

However, the most popular programming languages of today, such as C++ or Pascal, do not support persistence. Although it is in principle possible to produce specially tailored translation systems for these languages in order to

allow them to have full persistence in the sense described above, the development and marketing of new language translation systems is very expensive.

There are two principle difficulties in providing full persistence for such languages.

1. No support for automatic storage recovery, (garbage collection), is provided in such languages. Instead the programmer must provide explicit calls on a de-allocation procedure to free memory. Whilst the basic technique of garbage collection is well known [Bishop77][Almes80] it relies upon the language run-time system being able to reliably find which memory locations hold pointers to other locations. This, in the state of the art, requires either special purpose computers in which memory words are tagged to distinguish pointers from non pointers, or if standard computers are used, it requires a regular and disciplined placement of pointers in the computers memory. Popular techniques are to ensure that all of the pointers in an object are adjacent to one another at the start of the object, and are preceded by a word which encodes the number of pointers that an object contains.

   Popular programming languages like C, C++ or Pascal, allow the prorgammer complete freedom in deciding where pointers are to be stored within a data-structure, hence precluding the regular organisation needed for a garbage collector.

2. In object oriented programming, data objects commonly contain pointers to a data structure called a virtual method table. The virtual method table is a table of procedures that are permited to operate on the object. A problem with providing persistence for languages like C++ and Object Pascal is that the machine address at which this virtual method table resides may change between successive versions or runs of a program. Thus objects created at an earlier time may have invalid pointers to virtual method tables. This precludes simply copying objects in from disk when they are needed.

Two well known additional issues involved with persistent programming are that:

1. A large virtual address space may cause disk thrashing to occur when space is being allocated.

2. A mechanism is required to ensure that the store can be rolled back to a correct state following an error.

The persistence management system that we have implemented has features designed to overcome both of these.

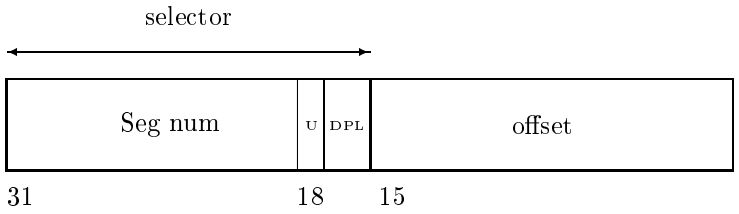

Figure 1: The format of a 32 bit segmented intel address. Note that the DPL bits control access rights so that there are only 2^{30} distinct addresses.

1.1 The address space architecture

CPUs derived from the Intel 386 architecture support two different virtual memory mechanisms. At the base level the machines support a 32 bit linear address space, on which a segmented address space, of either 46 or 30 bits, is overlain. In version 3.1 of the Microsoft Windows operating system, user programs saw only the upper layer, giving them a theoretical virtual address space of 1 gigabyte. Windows NT and Windows 95 allow programs to be compiled to directly access the 32 bit linear space, and moreover, they provide a library of routines that can be used to map ranges of linear addresses onto disk files. These new facilities could be used to provide the infrastructure of a persistence system. The new programming model is, however, poorly supported by some popular program development tools which continue to produce code modules designed to run in the 30bit segmented address space, which provides a strong incentive to develop a persistence manager that will run in this context.

The restrictions involved in sticking to the segmented model are not as severe as might be supposed. Windows 95 restricts new model programs to the lower 2 Gigabyte of linear memory whilst reserving the next Gigabyte to segmented memory applications, and reserving the top Gigabyte to the operating system. There is thus only a doubling in total address space entailed in moving to the linear addressed model. More serious is the absence of explicit support in the operating system for memory mapped files.

Use of the virtual memory for persistence involves being able to detect when an access is made to a data structure currently on disk so that it can be brought into RAM before the program continues. In terms of control over system facilities this demands abilities to:

1. capture the 'not present' interrupt,
2. set up the tables used by the virtual memory hardware
3. reserve areas of random access memory for active data,
4. reserve areas of disk for passive data.

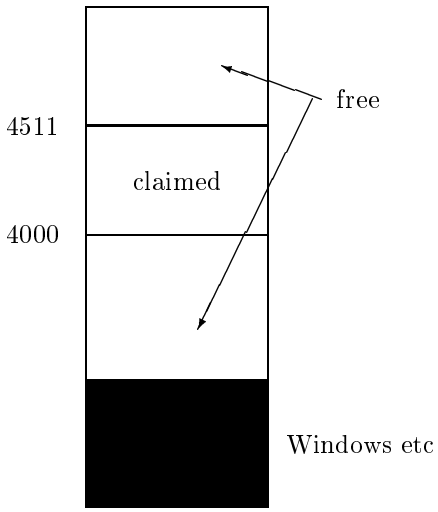

Figure 2: The segment map after running the claim algorithm.

Fortunately, windows is built on top of the Dos Protected Memory Interface (DPMI) which provides access to all of the required facilites.

2 Virtual Memory for persistent store

If one is to use virtual addresses for persistence, an advantageous technique is to reserve a range of virtual addresses that will constitute the persistent store. There exists no explicit system call to do this under DPMI, but there does exist a call to claim an individual segment selector. The effect of claiming a range of segments can be had by the following algorithm. Suppose we wish to claim 512 segments (32 megabytes) starting at segment 4000, then we:

1. Call the allocate selector procedure 4512 times, recording in a bitmap which selectors were allocated. Since some n selectors will already have been reserved by other applications before our code starts this will have reserved all segments up to $4511 + n$.

2. Check that all segments between 4000 and 4511 have been reserved. If not free all the claimed selectors and abort.

3. Free all the segments that were claimed below 4000 and above 4511.

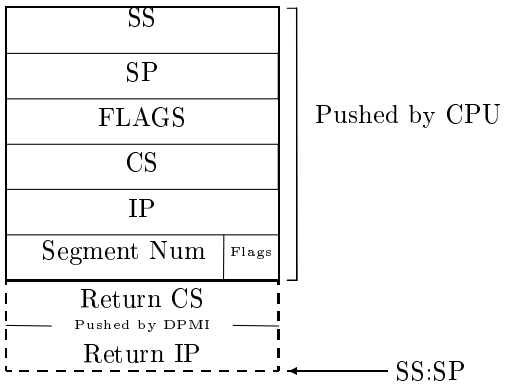

Figure 3: The configuration of the stack on entry to the segment not present exception handler.

2.1 Segment fault handler

Once an area of address space has been reserved for persistent applications, we then need override the DPMI default event handler to ensure that we can swap segments to and from disk on demand.

To each selector, there corresponds an entry in a table of descriptors. The descriptors are data structures which specify the base address, size and other attributes of the segment. One of these attributes is a present flag which is checked each time a segment register is loaded. Segment not-present exceptions are triggered when an application loads a segment register with a selector corresponding to a segment that is marked as not present.

When a not-present exception occurs, the CPU saves the context of the faulting code on the stack and pushes an error code which indicates which segment caused the fault, and under what circumstances. The exception is first handled by the operating system which then forwards it to the users exception handler. The resulting stack configuration is shown in figure 3. This configuration is unsuitable for a call to a high level language procedure, in that the CPU has only pushed the minimum state required to restart the application. The on entry to the handler most of the CPU registers contain the values that they had at the moment the fault occured, and there is every probability that these would be corrupted by a Pascal or C procedure. The first level fault handler has thus to be written in assembler to save the old registers, and to set up the data segment register correctly to ensure that the persistence manager will have access to its local variables.

The handler then checks to see if the segment which caused the fault is one

for which it is responsible. If not control is passed back to the operating system to handle it. Otherwise, a call is made on the second level fault handler. This has to claim physical memory for the segment and load it from the disk file in which it is stored. The claiming of memory is done by the standard windows process of obtaining a handle to a global memory block of the appropriate size, fixing it, and obtaining the linear address of the block. The descriptor for the faulting selector is then set to point at this linear address, its limit is set to the size of the segment and the access rights set to present with read only access.

Any attempt to write to data in the segment causes a general protection fault (interrupt 13). By trapping this fault too, the object manager can selectively switch on write enable bits for segments that are modified. When segments are swapped back to disk, inspection of the write enable bit in the segment table allows one to determine if the segment has been modified and must be saved, or may simply be discarded.

2.2 Backing store organisation

Passive data resident on disk needs to be appropriately organised. It has to provide both an area of disk to store persistent data when swapped out, and some form of transactional security. One has to ensure that a consistent image can be read off disk even if an earlier program had crashed whilst running. In a system that swaps data between memories, there is always the chance that a crash may leave the version in non-volatile memory half completed, with some blocks updated and some in an earlier state.

In DPOS this transactionally secure disk store is provided cheaply and simply using the filing system. There is a reserved directory `\SEGS` used to store persistent data. The currently versions of segments are stored in files with the suffix `.SEG` and the previous version of a segment is stored in a file with suffix `.BAK`. Thus the most recent copy of segment 4001 would be in file `\SEGS\32015.SEG` and the backup copy, if any, in file `\SEGS\32015.BAK`.

In addition to the segments, the system stores housekeeping tables in the `ROOTBLOK` file, whose structure is shown in figure 10. It has 4 components:

garbroot This is the unique pointer from which all persistent data must eventually descend. The garbage collector uses it to determine which objects must persist. It will contain a pointer into a record in the persistent store.

sels This is a table containing flags used to describe the attributes of all of the selectors. These flags indicate if the selector's segment is reserved for the persistent store, if it is currently holding persistent data, and if the data it contains is executable instructions.

htb This is an array of heaps whose purpose will be described in more detail below.

```
procedure vec11handler(errcode:word);far ;assembler; { dpmi gp handler }
          label echo,handle,done;
 asm
     pusha             { save calling context }
     push es
     push ds
     mov ax, seg @data { set up data seg of handler }
     mov ds,ax
     mov ax,errcode    { check if persistent seg caused }
     shr ax,3          { the not present fault }
     cmp ax,firstper shr 3
     jl echo
     cmp ax,lastper shr 3
     jg echo
     jmp handle        { it did, we must handle it }
echo:push errcode      { pass it to the operating system }
     call echoit
     jmp done
handle:
     push errcode
     call fetchseg     { load the segment }
done:                  { restore calling context }
     pop ds
     pop es
     popa
     leave
     retf
 end;
```

Figure 4: The first level fault handler in assembler.

```
   twurzel= record
              garbroot:^tpdata;
              sels:array[0..$1fff] of tselsort;
              htb:array[smallest..bigest] of theapid;
              classes:array[firstper div 8 .. lastper div 8] of pclassid;
            end;
```

Figure 5: The type of the root block which provides housekeeping tables for the persistence manager.

classes This is an array, indexed by persistent segment numbers, of pointers to class descriptors. These too, are described below.

When the persistence system is initialised, the ROOTBLOK is read into RAM and the time at which the file was last-written noted in a variable `timestamp`. The `sels` array is inspected and the selectors currently used by DPOS are identified. The following algorithm is then executed: For each segment in use

If its .SEG file was last written since the timestamp Then

erase the .SEG file

rename the .BAK file as the .SEG file

At the end of this, any segments that had been flushed to disk since the last checkpoint operation will have been replaced by their previous versions.

When a segment is flushed to disk the following occurs:

If no .SEG file exists Then

create the file

write the segment to it

Otherwise

If the .SEG file is newer than the timestamp Then

overwrite the existing .SEG file

Otherwise

erase any .BAK file

rename the .SEG file as a .BAK file

create a new .SEG file

write the segment to it

This ensures that any .SEG file that predates the timestamp is preserved in the event of a subsequent crash.

When the checkpoint procedure is invoked, all segments in memory are flushed to disk followed by the ROOTBLOK file. This ensures that the timestamp on the ROOTBLOK will be later than those on all of the segments in the database.

3 The persistent heap

The persistence manager can be thought of as having 3 layers of software.

1. The segmented store manager, which handles the swapping of segments to and from disk and which provides primitives to allocate and extend persistent segments.

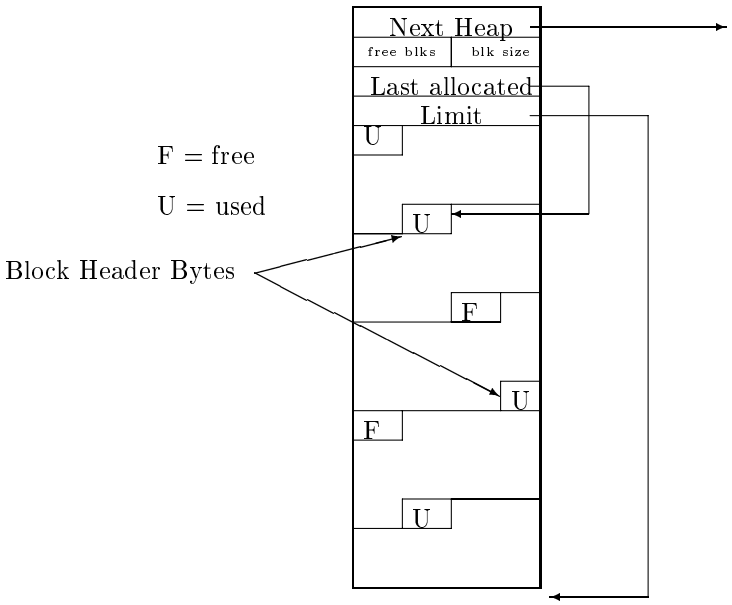

Figure 6: The organisation of the heap within a data segment. Note tha all heap blocks are of the same length.

2. The heap manager, which provides a multiplicity of persistent heaps and a garbage collector operating over them.

3. The class manager, which is responsible for the allocation of intsances of persistent classes and ensuring that their methods are reachable.

The heap manager has to allocate and recover space for a large number of objects, of differing but generally small sizes. It would clearly be unwise to allocate whole segments to individual objects, as this would only permit the creation of a few thousand objects at the most. Instead, some scheme has to be divised to split up segments allowing each one to hold a number of logical objects. Object allocation has to be reasonably rapid, and there has to and, where possible, it should occur without triggering excessive segment swapping.

The approach taken has been to create a multiplicity of heaps, each of which occupies one segment. Within a heap, space is organised as a sequence of heap blocks of uniform size. For example figure 6 shows a heap made up of 13 byte heap blocks, each of which contains 12 bytes of usable space. Each block has a header byte which contains 3 flag bits. These indicate if a block is free or used, marked or unmarked, a leaf node or one containing pointers.

Since all blocks are the same size, no freelist is needed, instead, a pointer indicates the last block to have been allocated. Allocation proceeds as a cyclical search of the heap to find the next unused block. Should none be found, the heap is extended by growing the segment that contains it. Growth occurs according to the golden ratio $r = \frac{a}{b} = \frac{a+b}{a}$. When the heap can grow no more, because of the 64K segment limit, a garbage collection is initiated. The unless the proportion of free space in the heap becomes very low, the cyclical search for a free block only has to visit a few block headers. The uniformity of block sizes within the heap means that the algorithm does not have to waste time visiting any nodes that are too small. The restriction of a heap to the supply of blocks of a single size, implies that there must be multiple heaps available, if a range of object sizes is to be supported. Since one can not afford to have a unique heap for each possible object size, a compromise is reached whereby each distinct object class is assigned a heap whose storeage blocks are just large enough for the instances of the class. This leaves those data objects that do not belong to definite classes, basically arrays of various sizes. These are handled by a vector of heaps with ascending block sizes. Arrays are allocated from the heap with the smallest practicable block size. The overall arrangement is seen in figure 7.

Since the number of objects required for a class or a given range of array sizes may exceed what can reside in one heap, each class or array range can potentially have a list of heaps. The existence of lists of heaps could lead to delays and thrashing when allocating new objects. An attempt is made to obviate this by providing two pointers to the list. One points at the head of the list. A second, points at the heap currently being used for allocation. Thus if the first three heaps on the list were full, the current allocation might come from the fourth. This avoids visiting the full heaps each time an allocation occurs.

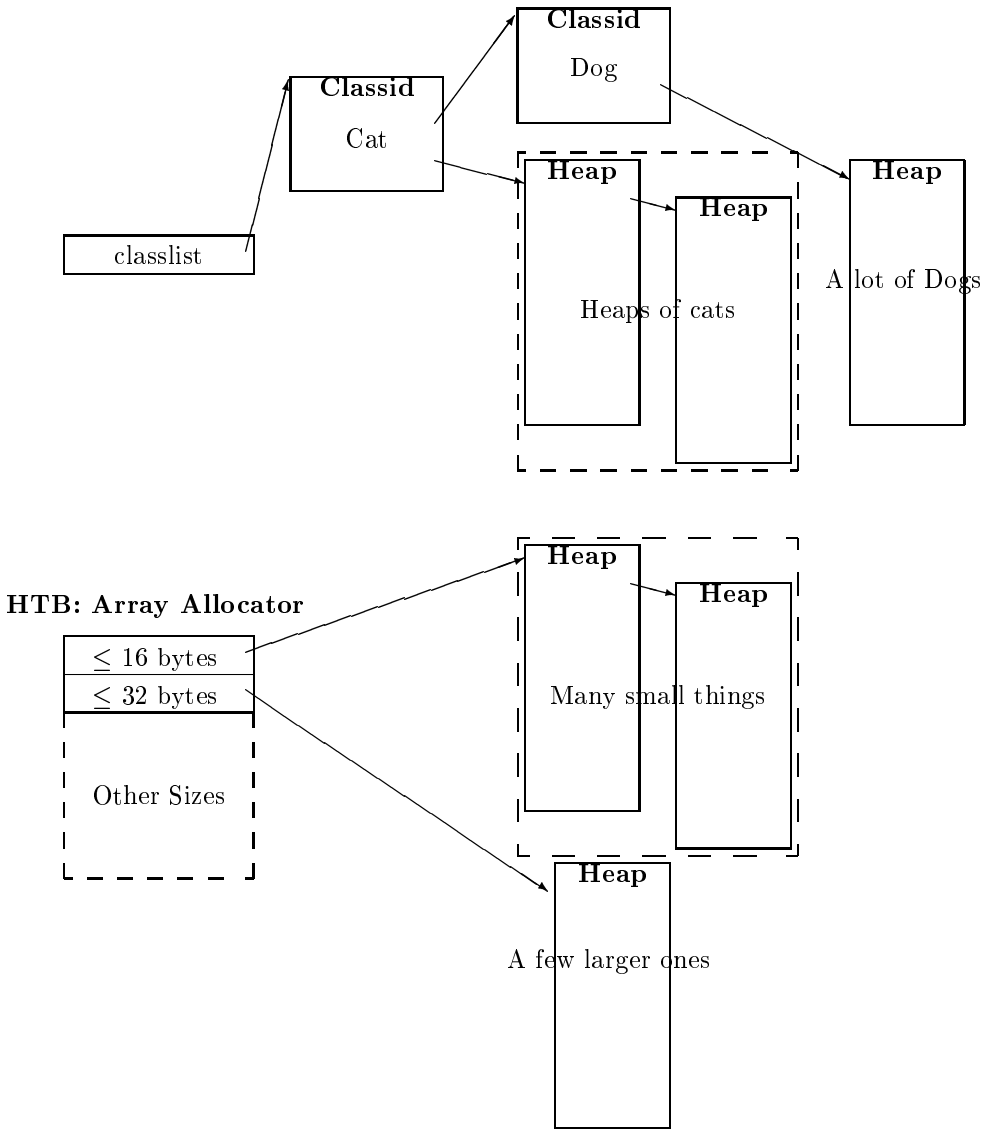

Figure 7: The storage allocation scheme uses multiple heaps. Each class has one or more heaps. Distinct heaps are provided for ranges of array sizes.

4 A garbage collector

The storage organisation desribed above has the features required to allow garbage collection of languages like C or pascal. The key step consists firstly of recognising that, on a computer with N bit addressing, valid addresses for objects comprise a subset of the the N bit integers. If some algorithm can be devised to computationally distinguish between those integers that could and those that could not be valid addresses, then, there is no need to tag or otherwise extraneously distinguish pointers from other machine words. Secondly, it consists of devising an organisation of objects in computer memory, such that a couple of table lookup operations can determine if an arbitrary N bit integer is in fact a valid object address.The key features are:

1. Allocating one or more ranges of addresses to be used to hold persistent objects. Call these ranges heaps.

2. Constructing a table indexed on high order address bits indicating if a given address falls within a heap.

3. Ensuring that all valid objects within a given heap occur at multiples of a heap-specific stride through the address space.

4. Placing immediately before the start of each object a marker byte that can be used to distinguish allocated from unallocated memory blocks.

Thus the table described in point 2 can be used to determine if the upper bits of a 32 bit integer are a valid address, whilst the information provided in points 3 and 4 can be used to check the lower bits. It should be noted that the use of steps 1 to 3 would itself be sufficient to create a reliable garbage collector - one that will not delete any object that ought to be retained. Step 4 is added for efficiency reasons only, to prevent the garbage collector wasting time scanning blocks of memory that have not currently been allocated.

The garbage collector proceeds as follows. It has access to the stack and data segments of the program, along, possibly, with one or more distinguished pointers that are designated roots of persistence. It scans the stack and data segment of the computer and for each memory location, it tests to see if the word at that location is

1. Within one of the heaps.

2. If it is, whether it corresponds to the starting point of an object in the heap.

It can do this because the start and end addresses of all of the heaps are known to it, and, within any one heap all items start at multiples of a known block-size from the start of the heap.

Having found an object it is marked, using a byte reserved for that purpose at an address one less than the starting point of the object, and the above scanning procedure is applied recursively to the internals of the object. The maximum possible size of the object is known to the garbage collector because of the known blocksizes of the heaps.

Once the objects are found, what we have is a conventional mark and sweep garbbage collector.

5 Pointers to virtual method tables

Object oriented languages like C++ and Object Pascal support what are called *virtual methods*. Unlike plain vanilla methods, whose procedure addresses can be computed at compile time, the address of a virtual method may not be known until run time. Consider the case of an object class `number` that declares a virtual method `show`. The method `show` is assumed to print the number as floating point decimal. Suppose we now declare a class `rational` as a subclass of `number` with a `show` method that prints numbers out as a fraction.

Given an instance `x` of the class `number`, the command `x.show` must invoke a different method according to whether `x` is a plain old number, or is a rational number. The solution conventionally adopted to this problem is shown in figure 8. The objects themselves have a pointer, either full length, or datasegment-relative, to a table, the Virtual Method Table, that contains pointers to the actual methods used. There will be different Virtual Method Tables for plain `numbers` and for `rationals`, but in each of these tables, the entry that points to the `show` method will be at a position that is known at compile time.

If however these two classes are incorporated as libraries into different programs, it is likely that the addresses of the virtual method tables for the class `rational` in the datasegment will differ between the programs. Thus if a rational number is created in the persistent memory by one progam, but is subsequently used by a second program, the virtual method table pointer will be incorrect on the second occasion. Our implementation obviates this problem by a two step process:

1. Each class that is used in a program must be registered with the persistent store before any instances are created or used. This is done by passing

   - a suitable identifier for the class, for example a string containing its name, along with
   - the size of the instances of the class and
   - the size of the virtual method table pointer
   - the position within the object of the virtual method table pointer

   to a registration procedure.

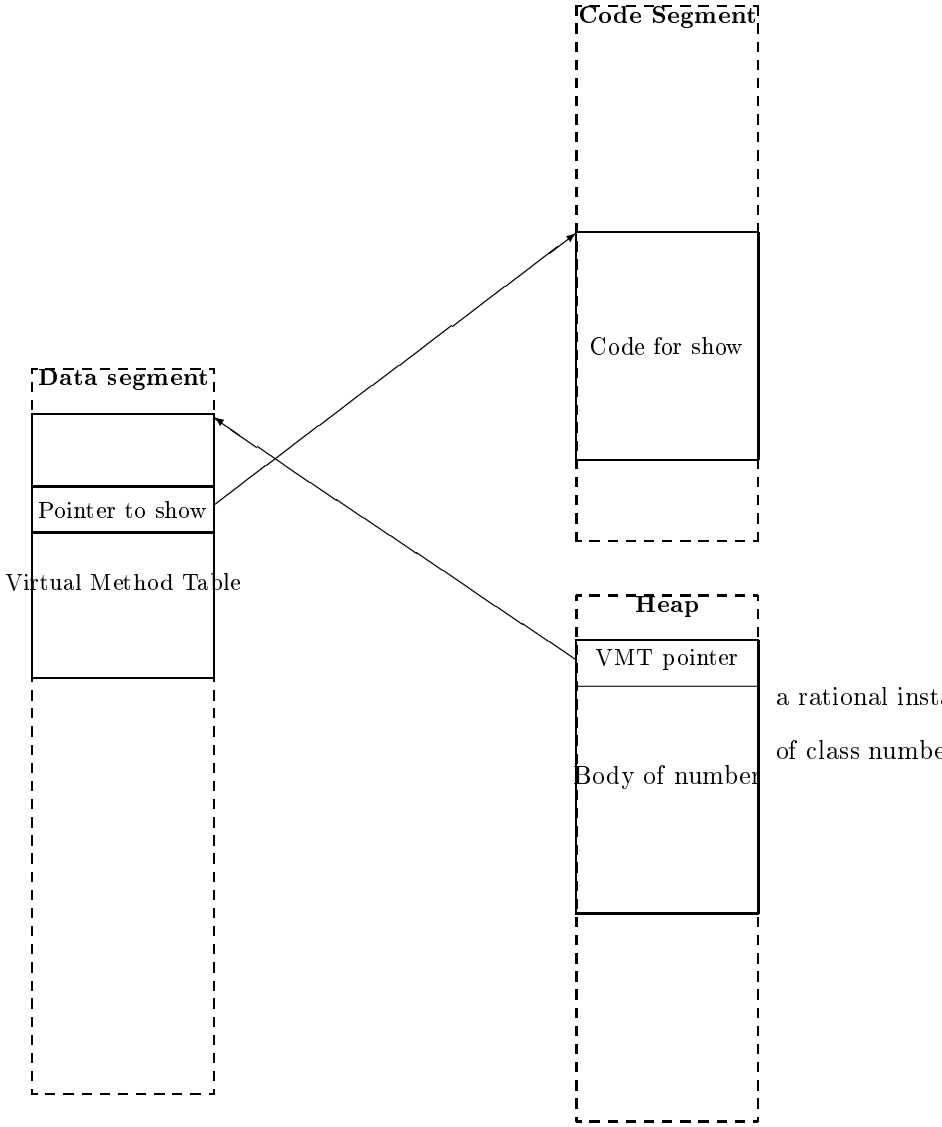

Figure 8: The use of a virtual method table in object oriented programming

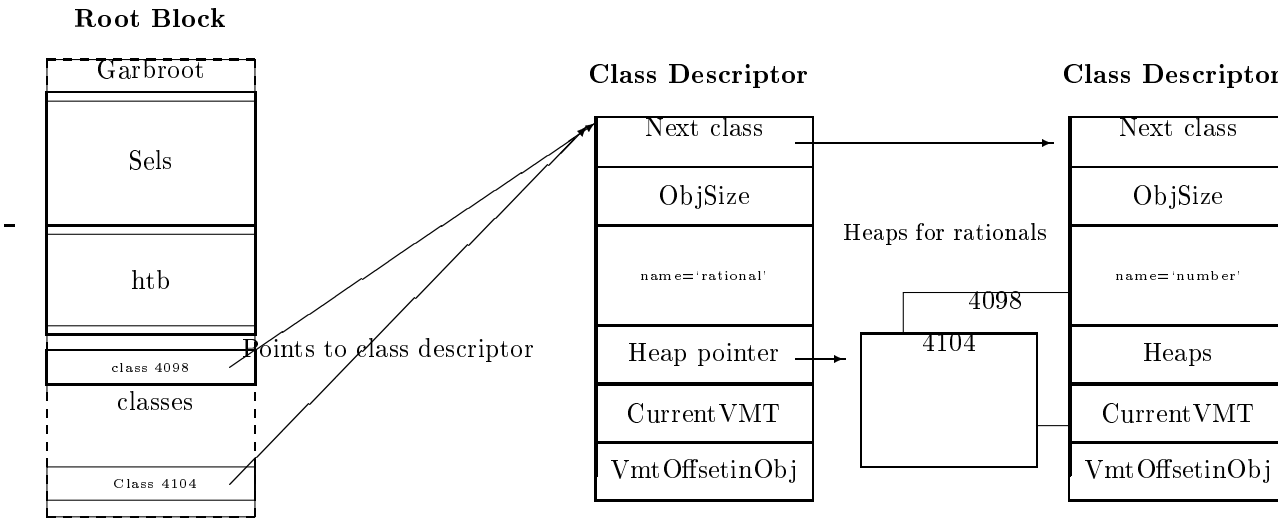

Figure 9: The heaps in segments 4098 and 4104 are show assigned to the class of rationals. On a segment fault, it is possible by inspecting the classes table to find which class 'owns' a given segment.

This procedure then creates a heap specially tailored to that class, and creates a class descriptor for the class if none exists in the persistent store at that time. The class descriptor has the form shown in figure 9.

An entry is then made into the **classes** table at the index position of the persistent segment used for the heap, pointing to the class descriptor. If in the course of time, this heap is insufficient to hold all of the members of the class, additional ones are created. All the heaps owned by a class have their entries in the classes table set to point at the appropriate class descriptor.

The system stores housekeeping tables in the `ROOTBLOK` file, whose structure is shown in figure 10. It has 4 components:

garbroot This is the unique pointer from which all persistent data must eventually descend. The garbage collector uses it to determine which objects must persist. It will contain a pointer into a record in the persistent store.

sels This is a table containing flags used to describe the attributes of all of the selectors. These flags indicate if the selector's segment is reserved for the persistent store, if it is currently holding persistent data, and if the data it contains is executable instructions.

```
twurzel= record
           garbroot:^tpdata;
           sels:array[0..$1fff] of tselsort;
           htb:array[smallest..bigest] of theapid;
           classes:array[firstper div 8 .. lastper div 8] of pclassid;
         end;
```

Figure 10: The type of the root block which provides housekeeping tables for the persistence manager.

htb This is an array of heaps whose purpose wax described in more detail above.

classes This is an array, indexed by persistent segment numbers, of pointers to class descriptors. These too, are described below.

2. When an address fault occurs, the hardware passes the selector of the faulting segment to the interrupt handler. From this can be derived the segment number, allowing the classes table to be inspected. Should the entry in the classes table corresponding to the segment number be non-nil the descriptor of the class owning that segment is obtained.

   From the class descriptor, the value and offset within the instances, of the class's current virtual method table may be determined. This information can then be used to update all of the instances of the class, resident within the faulting heap segment, to point at the currently valid virtual method table.

   By the time the interrupt handler returns, the instance of the class whose pointer caused the fault has been loaded into memory, and its virtual method table made accessible.

6 Self installation

In order to make persistence transparent, it is necessary to over-ride the built in heap mechanism of Pascal. Syntactically the interface to the Pascal heap is provided by two 'procedures': `new`, and `dispose`. These differ from other procedures in not being strictly type checked. One can pass to `new` a pointer to any type, and it will create a new instance of the type. This violation of the type rules is an indication that these procedures are not what they seem to be. In fact they are not procedures at all, but a piece of 'procedure like' disguising what the compiler really generates.

A moments thought will reveal that for `new` to work it really requires two parameters: a var parameter for the pointer to be returned and a second hidden

parameter specifying either the size or the type of the object being created. The Borland Pascal compiler makes the underlying procedure directly available as:

```
procedure GetMem(var P: Pointer; Size:  Word);
```

In this form it is useful for creating arrays on the heap whose size is not known until run time. `Dispose` is similarly translated by the compiler into a call on `FreeMem`.

During its startup sequence the persistence manager replaces the first instruction of `GetMem` with a jump to its own store allocator. The first instruction of `FreeMem` is replaced with a return instruction, since in a system with a garbage collector, freeing memory is a null action.

The extended syntax that is used with constructors `new(p,init(a,b,...))` is translated by the compiler directly into a call on the user declared constructor `init`. At the start of the constructor method the compiler plants a call to another store allocation routine , call it `PrivateAlloc`, which, instead of being passed the objects size, is passed its VMT offset. Since the first word in a VMT always contains the size of the object, this provides sufficient information to create the object.

The persistence manager patches `PrivateAlloc` to branch to an allocation routine that searches the list of persistent classes for the one with the appropriate VMT and then allocates an object from the private heap of that class. For this to function, the class list must have been updated to have entries for all persistent classes. Thus as part of its initialisation sequence, the persistence manager searches the data segment for VMTs and registers all the persistent classes whose VMTs it finds.

References

[Aho80] Aho,A. and Ullman,J. "Optimal partial-match retrieval when fields are independently specified" ACM Transactions on Database Systems, 4:2, 1979, 168-179.

[Almes80] Almes, G. T., "Garbage collection in an object oriented system", PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1980

[Atkinson83] Atkinson, M. P., Bailey, P., Chisholm, K.J., Cockshott, W. P., Morrison, R., "An approach to persistent programming ", *The Computer Journal*, Nov 1983, 26, 4, 360-365.

[Bishop77] Bishop, P. B., "Computer systems with a very large address space and garbage collection", PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1977.

[Bitton 83] Bitton e. al,Benchmarking Database Systems - a systematic approach,in proceedings VLDB 1983

[Cockshott83] Cockshott, W. P., "Orthogonal Persistence", Ph.D. Thesis, University of Edinburgh, 1983.

[Cockshott85] Cockshott, W. P, "Addressing mechanisms and persistent programming", in *Persistence and datatypes*, Papers for the Appin Workshop August, 1985, Persistent Programming Research Report 16, University of Glasgow Department of Computer Science.

[Cockshott 96] W. P., Paul Cockshott, John Gilchrist,Douglas McGreggor, Phil Murray, John Wilson "The HIBASE Compressed Database", Department of Computer Science, University of Strathclyde 1996.

[Garcia92] Garcia-Molina,H. and Salem,K. "Main memory database systems: an overview" IEEE Transactions on Knowledge and Data Engineering, 4:6, 1992, pp509-516.

[Goldstein70] Goldstein,R. and Strnad,A. "The MacAIMS data management system" Proceedings of the ACM SCIFIDET Workshop on Data Description and Access, 1970.

[Lloyd80] Lloyd,J. "Optimal partial-match retrieval" BIT, 20, 1980, 406-413.

[Pucheral90] Pucheral,P. Thevenin,J. and Valduriez,P. "Efficient main memory data management using DBGraph storage model" Proceedings of the 16th VLDB Conference, Brisbane 1990, pp683-695.

[Rivest 76] Rivest,R. "Partial-match retrieval algorithms" SIAM Journal of Computing, 5:1, 1976, pp19-50.

[Wang 89] Wang,C. and Lavington,S. "The lexical token converter - a hogh performance associative dictionary for large knowledge bases" Department of Computer Science, University of Essex, Internal Report CSM-133.

[Suzuki94] Suzuki, S., Kitsuregawa, M., Takagi, M., "An Efficient Pointer Swizzling Method for Navigation Intensive Applications", in *Persistent Object Systems*, Atkinson, M., Maier, D., Benzaken, V., Eds, Springer Verlag, 1994.