

Box 3
Folder 10

Virtual Memory Interest Group, 1976-1980

Adele Goldberg papers

Smalltalk

X5774.2010, Box 3

102739346

THE COMPUTER HISTORY MUSEUM



1 027 3934 6

Virtual Memory

Oct, 15, 1979

goals

what does it mean to be "sharing"? is it a
"read-only" concept or are notions of
locking needed, etc.

copying w/ sharing to save space only
true sharing — multiple views across users

Bobrow + Goldstein

access w/ context (layers)
composition of views $\Rightarrow$ view

ind'l objects being shared — when have the
"same" thing

Time coordination for seeing a change
of a shared object

what viewing method is now appropriate?

network address space

problem: single wizard write on the community store
: reading — ~~data~~ dynamic loading
application & package release

name spaces
pointers
converting from
name to an object
writer
binding
time for
name space } different binding properties

versions / labelling
theory of forgetting

sharing parts

not actual sharing

but constrained 3 need to establish path

msg passing purpose ~~is~~ is to let an
object appear as though it has
virtual parts —

if hand out pointer to part, can't change
your mind later about making your
part be virtual (represented diff'tly)

can have ptr only if ptr is really
a descriptor not real thing so can't any changes

Juniper paper: multiple users of logically remote object

- phantom
- grain too tight $\rightarrow$ local cache
see Lamport's algorithm
idea of taking control to
~~one~~ user at a time

next mtg { next week - Ira & Danny
" - Austin

XEROX
PALO ALTO RESEARCH CENTER
Learning Research Group
October 6, 1979

From: Dan Ingalls
To: Virtual Memory Interest
Subject: Making Space for Smalltalk
Filed On: MAXC<INGALLS>Vmem.Bravo

During the coming year, LRG will be building a new virtual memory for Smalltalk. Our plan is to spend about a month specifying design goals, two to three months completing a design, and about four months on implementation. This memo began as a talk with Alan, and its purpose is to present a set of goals as a target for criticism and a stimulus to contributions during the first phase.

Many Users on Different Machines

From the beginning, we want to provide for many users sharing the same large environment on separate machines. We feel that the assumption of ethernet-like communication forces an appropriate modularity to the design. Moreover, we want to assume that the various Smalltalk users may be running on different interpreters in machines of differing types. If done right, a NoteTaker user should be able to execute a save command, and then resume that saved image on a Dorado. Moreover, it should be possible to spawn processes which could migrate to any machine on the ether and continue their simulation, with the apparent effect of adding processing capability to the shared database, yet with no need to alter the real underlying database server.

Shared Archival Data and Workingset Management

The design issues fall into two fairly distinct groups: those that deal with the management of a large shared archival database, and those that deal with management of a small local unshared workspace. Rather than spend the next year trying to figure out how these two are the same, we shall accept this dichotomy and try to turn it to our advantage. To the extent that the archival and sharing issues are independent of the local workspace issues, they should be able to be handled at a high level in Smalltalk, and we may be spared the drain of separate implementations for differing target machines. Similarly the local workspace issues may be tailored to each machine - NoteTakers which detach for remote use, Dorados which have hardware paging support, and so on.

Swapping Behavior

We have learned from Smalltalk-76 that object swapping really works - even in the face of significant performance bugs. However, besides removing those bugs, we would like the next virtual memory to take advantage of the connectivity of typical workspaces. For instance, if an object is to be purged, it makes sense that all other resident objects that are only pointed to by it be also written out at the same time, and to the same place. We should be able to take even greater advantage of this locality of reference by incorporating some of our earlier ideas into the structures of both the local workingset and the archival data.

Relationship to Existing Systems

With the vast resources at our disposal, it should be simple to implement our own Juniper and design another Smalltalk interpreter from the ground up. However, just to be on the safe side, a figure of merit for our design will be its ability to work with exactly what is currently available for archival data management and interpretation of Smalltalk. Juniper and possibly an IFS-based B-tree server are the most likely choices for the former, and the existing NoteTaker interpreter, with its carbon-copy Dorado and Alto brothers is the only choice for the latter.

Food for Thought

Many issues, each worthy of PhD work, spring immediately to mind:

How should the shared database be addressed?

What should be the structure of remote objects?

How should both the shared and local spaces be partitioned to provide swapping units which are neither too large nor too small?

When an external object is brought into a local space, should it be "exploded" (ie, all its parts relocated relative to that space), or should it constitute its own "zone" as a virtual machine within the host?

What does it mean to "release a new system" in this context?

How do we ensure that a failure of one machine does not compromise the integrity of the entire system?

-- Messages from file: [PARC-MAXC]<ADELE>MESSAGE.TXT;1
-- MONDAY, OCTOBER 15, 1979 10:15:56-PDT --

Date: 12 Oct 1979 2:37 pm (Friday)
From: Marshall.WBST
Subject: Re: Smalltalk Virtual Memory meeting
In-reply-to: Your message of 10 Oct 1979 12:25 pm (Wednesday)
To: Ingalls.PA
cc: MBrown.PA, McJones.PA, Lampson.PA, Masinter.PA, Kay.PA, Marshall,
Weyer.PA, McCall.PA, Adele.PA, Alan.PA, Baldwin.PA, Strollo.PA, Fikes.PA,
Robson.PA, Kaehler.PA, Krasner.PA, Tesler.PA, Bobrow.PA, DaveSmith.PA,
AHenderson.PA, Lansford.EOS, Goldstein.PA, ABell.PA, Horn.PA, Althoff.PA,
Deutsch.PA

Although I won't be at the virtual memory meeting I would like to contribute the following ideas. I apologize for the length of this message.

The current Smalltalk-76 uses reference counts at all times - even when accessing its local stack. The paper by Deutsch and Bobrow shows how to eliminate reference counting stack references at the expense of a stack scan when garbage collecting. I would like to generalize this concept by dividing all objects into the two sets "active" and "inactive". (Think of active objects in RAM and inactive objects on disk although this doesn't have to be true.) Reference counts would only count references of pointers in inactive objects. Pointers of active objects would not be counted. If pointers are only changed in active objects then no pointer manipulations would have any reference counting overhead. Reference counts would only be modified when an object changed from inactive to active or vice versa.

To keep track of reference counts a hash table can be kept. Inactive objects with refct not equal to 1 and active objects with refct not equal to 0 must be included. This hash table might be split into several tables [e.g., ZRT and MRT] for efficiency.

To make an inactive object active one decrements the refct of all objects pointed to by the object's pointers. To make an active object inactive one increments the refct of all objects pointed to by the object's pointers.

Reclamation is more difficult. Basically, a tracing garbage collector is needed for active objects. Starting at 'Smalltalk' and the current context (or whatever) and all active objects with nonzero refct objects are marked. Marking terminates on inactive or marked objects. Any object that is not marked can be deleted (Since it is active, no reference counts have to be changed). Any inactive objects that have refct of 0 and were not touched during marking can also be reclaimed by making it active (decrementing refcts and possibly providing more candidates for reclaiming) and deleting it.

Quite a bit of information is around during reclamation which could be used to optimize the swapping strategy. Objects with refcts of 1 should be placed near their owners. Objects with no active references could probably be made inactive etc.

The number of active objects should be small to speed up garbage collection and large to minimize faulting.

Advantages:

No reference counting during normal processing.
Circular structures recovered if all active during GC.
Useful info available for purging and allocation strategies.

Disadvantages:

Size of refct tables needed for inactive objects may be too big.
Full core trace needed to reclaim objects (but not to purge).

Possible extension:

I have divided objects into active and inactive. Maybe they could be divided into level-0, level-1, level-2 etc. with each level somewhat independent in its storage management and reclamation. Higher levels probably only have to worry about changing levels of objects.

To: Larry Tesler

From: Adele

Re: Virtual Memory and me

Sorry for the delay in reporting back, but here is a brief outline of some of my Smalltalk information sharing needs.

The main application I am working on right now is the Leesburg field study. In the long run I am interested in implementing a management environment for computer-assisted learning (the Ideas System). Neil and I have written several descriptive memos about the database for the Leesburg project and associated lessons; the first memo outlines the Ideas system as well. I have attached copies in case you want to scan them. Here I will simply outline the aspects of the applications programs that require timely access to the shared information. Each involves multiple users. Most can tolerate update periodically; exceptions are noted.

1. Creation and maintenance of a Xerox product and Xerox competitor's product database. Data access will be done (is experimentally now done) via browsers and forms of various kinds. A product description is a set of dictionaries plus some sets and strings; we are going to be revising this somewhat, but the point is the large number of large dictionaries. Therefore, unlike previous database experiences LRG has had, each primary object (instance of class Product) is very large, because each field is assigned to a large dictionary object. We expect to link in references to images of the products as well, but currently search a separate picture dictionary.

The current browsing methods use Findit to locate the product instance and then bring in the full object description. This is too space consuming and probably costly in time for lesson applications. We plan Smalltalk code for reading only the desired parts (fields) but would prefer some virtual memory assist--something like the following.

We write code to read the database, find it, and read the object in. Only it comes in as a "dummy," with parts assigned identifiers for the retrieval of the actual information. Only when code tries to use the parts does the actual data come in. This behavior is recursive in that once a product is retrieved, one of its dictionaries (parts) might be sent a message; but the whole dictionary should not be retrieved, only the entries that are referenced. I believe this is the obvious extension to Ooze and is already planned by the virtual memory design team.

This storage as dictionaries is going to change dramatically. We expect to redundantly store the information about products currently referred to as "needs, features, advantages"--the redundant store will be a topological one, much like a semantic net. The curriculum for each student will also be represented in this way in order to describe prerequisite and practice links. In both cases, the topology (a network) is to be stored separately with references to the models (needs, et al or lessons) because there are several possible topologies for any set of products or lessons. Bill Finzer did a first pass on this curriculum mapping problem last summer. It is harder than the product database representation because of the dynamic way in which it might change (see #5).

2. Lessons reading the database are described in both memos that I have attached. One of the research issues we are dealing with is this use of a database to generate lessons by including retrieval descriptions in the lesson models (we call these "unit models"). The database sharing problems are the same as #1, only with no update requirements.
3. Student history database maintained by output from the lessons and read by "teachers" for reporting. A single user would update a history (for that user), but a teacher might be browsing student histories at the same time. Since teachers are curriculum developers at, say, Leesburg, and students, ultimately, will be at branch offices (or, one day, at home), the difference in work hours will probably cause this conflict.
4. Territory management game described in the first memo will require multiple users updating a model of a sales territory. This is an example (similar to the ones I pointed out to Richard Moore) of a fixed model whose "parameters" are changed by multiple users; each user might require knowledge of changes made by the other users, especially if cooperative sales experiences are to be simulated.
5. The Ideas System includes a special use of a mail system in which a mail receiver is a machine on which the student/teacher histories are stored. The mail is from teacher to student history in order to modify the student curriculum instructions, that is the curriculum topology (see #1) (what the student sees when he or she selects "map"). This does not seem like a very unusual application, but includes the transaction problems caused by student reading while teacher is sending an update request.
6. The Ideas System includes the notion of unit models which represent templates for lessons for which curriculum writers might search, edit, and archive as new models or actual lessons. This is quite complex because a unit model includes an object we call a "guide" which links units. As a result, at the highest level of thinking about unit models, a unit is a curriculum for some subject area. So individual units are themselves nodes in a network. Multiple users might be searching the unit model database and multiple students might be retrieving the same lesson.

Using the Telephone

Some of the NoteTaker clients have database needs that are not unusual except for the requirement to use the telephone to gain access to the network and hence to the database. An example is to develop a database for an MRP (managing resource personnel, or something like that). The information stored is a simple record, but two fields can be lengthy commentary, much like the annotated bibliography that Steve Weyer once implemented. Typically retrievals would not require the entire record, only particular parts. Given transmission over the telephone lines and constraints on the NoteTaker memory size, we might consider retrieving only the actual record entries (much like the example #1).

Spying on the Current Implementation

Right now the kinds of applications we have implemented show that the code to retrieve a dictionary from the database represents only a small percentage of the execution time. Most of it is

in the fileserver interface and the etherworld code and some gains here would be much appreciated.

Inter-Office Memorandum

To Vmem Interest

Date November 7, 1979

From Diana Merry

Location Palo Alto

Subject Today's Meeting

Organization LRG

The meeting today included the restatement and discussion of goals. Most of the discussion was focused around the issue of sharing. What follows is an attempt to duplicate what remained on the board at the end of the meeting.

Mechanisms

Sharing -- how

Naming

Big address space --
Nesting -- address space

Grouping -- workset management

Sharing Code

- 1) Finding
- 2) Access on demand
- 3) Notification/confirmation/merging
- 4) Protected changes, trial execution
- Different ways of trying updates.
 - 1) Notify/confirm with checkpoint
 - 1a) Reverse action
 - 2) Versioning
- Several pivot points
- 3) Synchronization by version

Sharing Other Objects

Means

Locate common units
in different aspects of VM

Layers

- Versions
- Sharing
- Swapping
- Garbage collection
- Naming

Goals

Common x+1 VM
Compatible implementations
NoteTaker save/Dorado load
Inverse also

Release of Smalltalk and/or applications
Wizard says this is it
Pragmatic -- sharing all unaltered stuff
"Package" releasing
Protection -- crash survival
Garbage
Interface to SmallTalk '78

Transactions are a coarse grain on system world line

"Casting off" in a NoteTaker

Private objects

Provision for access

Interpreter

Core

Disk

Server

Beyond

Inter-Office Memorandum

To	Virtual Memory Interest Group	Date	April 23, 1980
From	Ted Kaehler and Dan Ingalls	Location	Palo Alto
Subject	Lambda Scheme for the Large Object Oriented Memory	Organization	LRG

XEROX

Filed on: IVY<Kaehler>LOOM.memo

This memo describes the winning design of the recent sweepstakes for the Large Object Oriented Memory (**LOOM**) to support Smalltalk 78. The design is called the lambda scheme and is a modification of the fully resolved pointer scheme. The virtual space is double word addressed by 31 bits, allowing a maximum size of eight billion bytes. The swapping unit is a single object (as in OOZE) and objects are packed in core. Objects in core are accessed through an object table (**OT**) which is indexed directly by 16 bit **local object pointers**. If a field of an object in core does not contain a local pointer, it contains an exception value called **lambda**. In an object in core, if a field containing a lambda is touched, the disk image of the parent object must be consulted to find out what pointer lambda stands for. Storage reclamation is handled with reference counts. This scheme is not tied to any specific strategy for grouping objects to lower the rate of faulting. Grouping may be accomplished by arranging objects on the disk into groups.

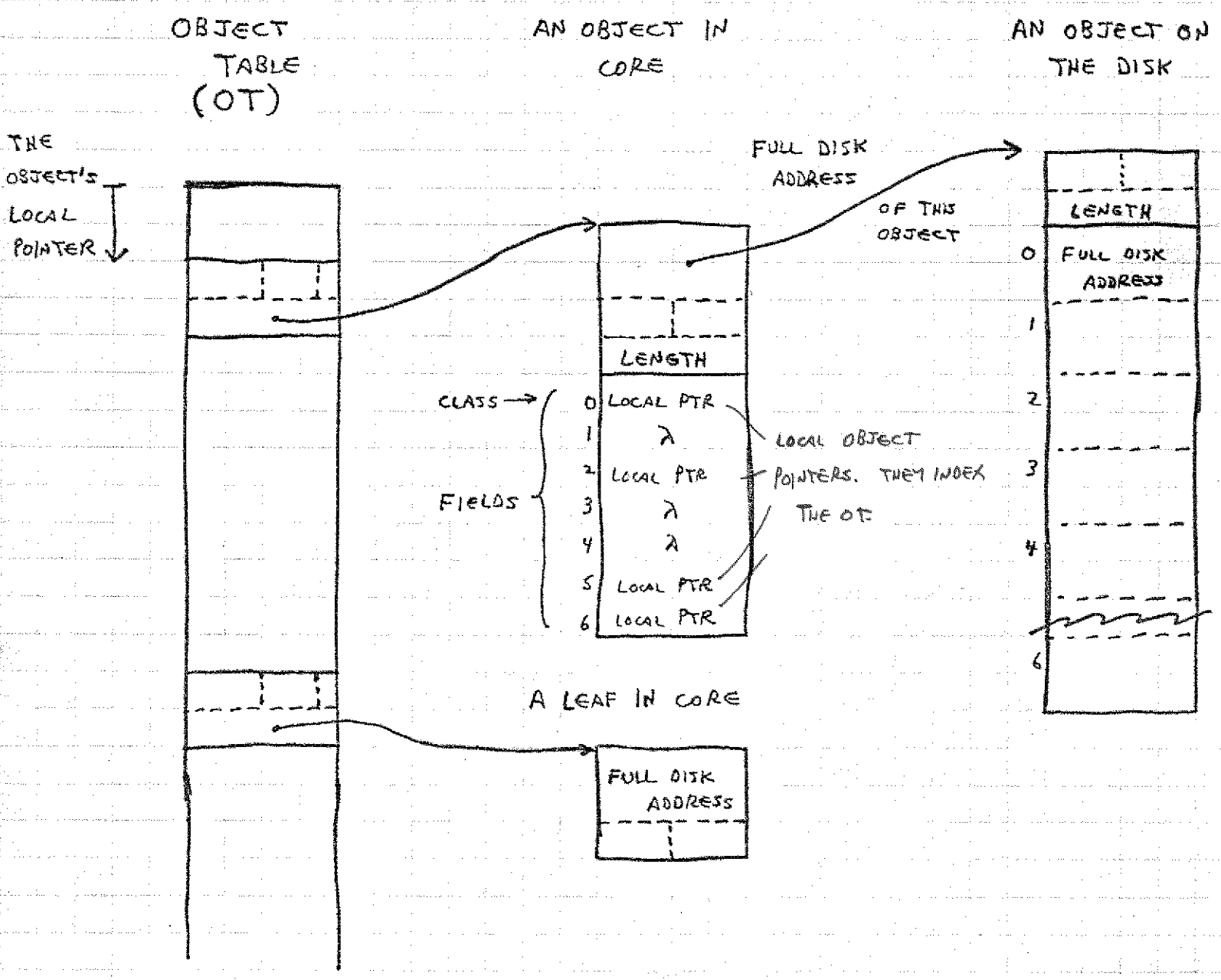
Credits

This design evolved as a true group creation. Jim Althoff and Steve Weyer turned our thinking to these lines away from strongly grouped schemes. Peter Deutsch produced the first version of the fully resolved pointer scheme. Dan, Glenn, and Ted worked out the reference counting scheme. Danny Bobrow recommended against giving a local pointer to every field that points to an object which is not in core. Dan and Ted explored the purging and file handling algorithms (while the scheme had two words per pointer). Larry Tesler contributed the lambda idea for unresolved fields from Caravan. Ted noticed that hints on the disk of which fields to resolve at fault time would greatly reduce the burden on the page buffers.

Pointer Manipulation in Core

A pointer field of an object in core contains 16 bits and is either a local pointer or a lambda. A local pointer is an index into the **OT**. **OT** entries are two words long and contain a reference count, some status bits, and the core address of the body of the object. (*See figure 1*). The header part of the object body carries the object's own disk pointer, its length, another reference count, and more control bits. A **leaf** is an object which has only its header in core and not its fields. (A leaf represents an object whose fields were recently thrown out of core, or a newly touched object with none of its fields yet touched). In a non-leaf, the fields follow the header. The pointer to the class of the object is not treated specially; it is just another field.

FIGURE 1. FORMAT OF AN OBJECT IN CORE



Every field that is loaded is tested to see if it contains a lambda. If it does, the disk image of the parent object is reread and a leaf is constructed to hold the full disk pointer from that field. The new local pointer is stored back over the lambda. Thus the Smalltalk interpreter sees only local pointers. If a lambda is stored over, it is first converted to a leaf as if it were being loaded.

Reclamation of Objects

Reference counts on objects determine when objects are no longer pointed to. There are three types of reference count. If all pointers to an object are in disk pointer format then the **disk reference count** of the object is its true count. This count never comes into core and lives in the image of the object on the disk. The **delta reference count** is the difference between the disk count and the number of disk format pointers which currently point to the object. Since the disk count is frozen on the disk, the delta count keeps track of changes as they occur. The delta count exists while the object is in core and is kept in the object header. The **OT reference count** is simply the number of fields in core that contain a local pointer to the object. The disk count plus the delta count equals the number of disk format pointers to the object. Since the OT count is the number of local format pointers, the sum of all three counts is the true reference count of the object. (See figure 2).

The OT count is the only one of the three that changes at high bandwidth. After a pointer has been followed once and converted to a local pointer, all stores of it or over it change only the OT count. When a fault brings an object in from the disk, its OT count is zero and its delta count is zero. The act of converting a lambda (which stands for a disk pointer) into a local pointer decrements its delta count and increments its OT count. The delta count is allowed to have negative values. Storing a new copy of a local pointer to an object increments the OT count.

The OT entry and the object itself have separate and unrelated lifetimes. When an OT count goes to zero, no more fields contain local pointers to the object. If the object does not have a disk pointer assigned to it (in its header), then it must never have been on the disk. It was a temporary object and may be reclaimed. If the object does have a home on the disk and its delta count is zero, then the disk count is already correct. Otherwise, the delta count must be added to the disk count before the OT entry and core may be reclaimed. If the disk count is zero after the delta count is added, then the object is truly not referenced and the disk space may be reclaimed.

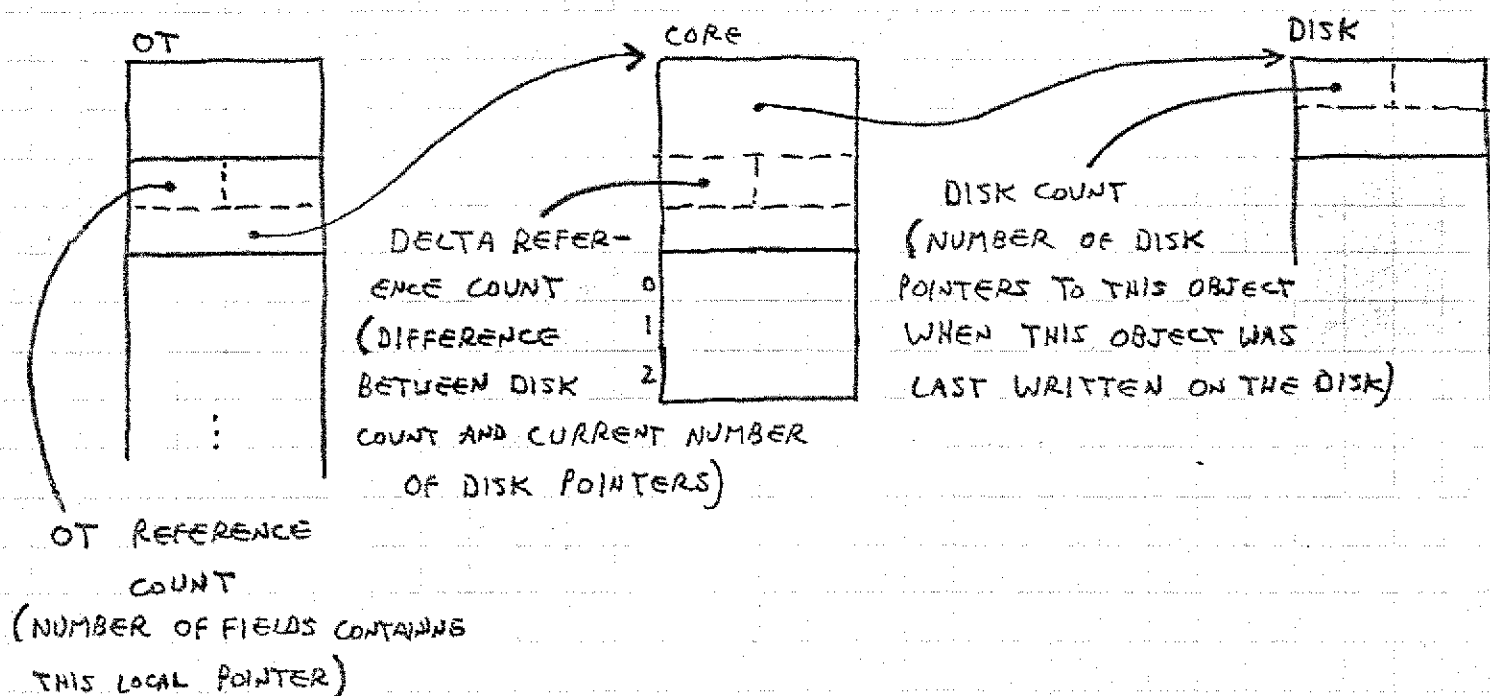
Faulting

The OT is a hash table with each object's disk pointer as its hash key. Each disk pointer key has a reprobe chain. The reprobe chain must be followed to the end to verify that a given object is not in core. This is because an object's entry in the OT cannot be moved once it is assigned. (Local pointers index to that location). The OT is kept somewhat sparse so that the reprobe chain can be short without excluding too many combinations of keys. The reprobe search has found the proper object when the current entry's disk pointer (from its header in core) equals the key.

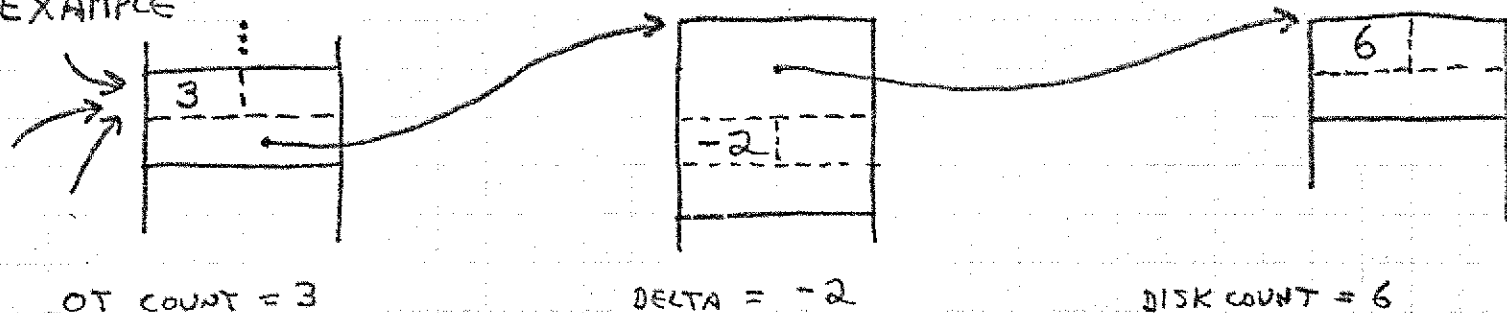
A fault begins when a field being loaded or stored over contains a lambda. The disk image of the parent object must be found and the field must be reread. The parent contains its own full disk pointer and also contains a hint of which disk buffer it last occupied. The hint takes the form of an index into the page buffer table (PBT) of pages in core. If the parent's page is no longer in the

FIGURE 2.

THREE TYPES OF REFERENCE COUNTS



EXAMPLE



SIX FULL DISK POINTERS EXISTED TO THIS OBJECT WHEN IT WAS LAST WRITTEN ON THE DISK. A DELTA OF -2 MEANS THAT TWO OF THEM ARE IN CORE AND HAVE BEEN CONVERTED TO LOCAL POINTERS. AN OT COUNT OF 3 MEANS THAT ONE NEW POINTER HAS BEEN CREATED TO THIS OBJECT BESIDES THE TWO CONVERTED DISK POINTERS. THE TOTAL NUMBER OF REFERENCES IS $3 - 2 + 6 = 7$. (IT COULD ALSO BE THAT 2 DISK POINTERS HAVE BEEN DESTROYED AND THREE NEW LOCAL POINTERS HAVE BEEN CREATED)

buffer, a fault on the parent occurs using the disk address. The field is reread, and its value hashed into the OT to see if the son is already in core. If not, a new OT entry and leaf are created for the son.

A second kind of fault occurs when a field is needed from an object which is currently only a leaf in core. (Asking for the class of the object in order to send it a message is common case). Once again, we use the page buffer hint in the object header to try to find the page in core. If the page is not in core, fault on the object using the disk address from the header. When the image of the object is in a page buffer, allocate enough core for the header and the fields. Each field on the disk has a bit which is on if this field was used the last time this object was in core. If the bit is on, create a leaf for that field. Put lambda in all the other fields.

A fault may need to find the page in secondary storage using the 31 bit disk address. A page number is extracted from the address. It is looked up in the page run table (PRT) to determine which device, file, and page number it is. The page is brought in and given a page buffer. Hash the page buffer table (PBT) with the page part of the disk address and install the new buffer.

Purging and Compaction

The creation of a leaf, expansion of a leaf to an object, and creation of a new object each ask the allocator for a block of core. When no core is available, a purge throws out objects to get more space. If, at the time an object is purged from core, its OT reference count is zero, it is simply written to its disk page. At the time of the write, overwrite only those fields which have local pointers. The fields which have lambdas in core remain unchanged on the disk. Convert local pointers to disk pointers by looking in the headers of those objects.

If the OT count of the object being purged is not zero, the fields are written out and the disk count changed to make the delta count be zero. The core occupied by the fields is freed and the object becomes a leaf. The leaf is clean and disappears when the last local pointer to it is converted to a disk pointer. The act of purging a normal object and leaving a leaf will evaporate other leaves as local pointers are converted to disk pointers.

Each object contains a touched bit and a dirty bit. These are set to the touched and clean values when the object is faulted on. Writing in the object dirties it. Not dirty and a delta count of zero signal a truly clean object. The strategy for deciding which objects to purge is to sweep core and to mark touched objects untouched. When an untouched object is encountered, it has remained untouched for an entire sweep, and deserves to be purged.

The order of objects in core is from oldest to youngest. Each sweep starts at a randomly chosen place in core and sweeps to the youngest object. Presumably old objects have gotten old by continuing to be touched. They are likely to remain in use. To avoid long searches of core which find no purgable objects, we sweep the old objects less often than the young ones. Young objects have just been faulted on. Some of them will be used for a short time and then never used again. We detect these objects early by sweeping their core often.

A sweep starts with an image of an object in core. The OT entry for that object is found by hashing its disk pointer into the OT. After the touched bit is inspected, use the length from the

object header to find the location of the next object in core.

Core is compacted by moving objects to new locations. Since the OT entry of any object may be found from its image in core, the core address of the object can be fixed up after it is moved. It is never necessary to sweep the OT and invert its pointers to core. Compaction may be done during the purge sweep to move objects up to fill holes. If compaction preserves the oldest to youngest ordering of core, all free core stays in a single block at the end. The inspection and moving of a single object during a sweep is a small atomic operation. Thus purging and compaction may be done incrementally during other computation.

Grouping and Moving Objects on the Disk

Grouping is accomplished by moving objects that are faulted on together to the same page on the disk. Perhaps we will fault on quad pages to allow more objects to come in on the same disk access. Many different grouping strategies may be tried. This design imposes no specific grouping strategy for objects. This is one of its prime advantages for our application, as it will be a superb testbed for research on grouping techniques. Initially, the virtual memory files will be grouped in a depth first traversal of the entire system. An offline utility program will statically regroup the files using our current best algorithm.

A simply way to regroup is to leave a touched and was-dirty bit on the disk for each object that was used. A switch in Smalltalk will cause resident objects to be written regardless of whether they are clean when purged. We will then run the system through its paces with the switch on and quit. The grouping program will divide the depth first ordering into untouched, readonly, and read-write areas. Segregating the untouched data will improve faulting buffer hits, and consolidating readonly objects will cause entire pages to be clean at purge time.

Versions, Page Sharing, and Safing

All disk requests go through the page run table (PRT) which directs access to the files which comprise the history of this version of Smalltalk. These files include one **current** file which is read/write and any number of readonly files. Attempts to write on the readonly files cause the page in question to be written out on the current file instead. The PRT must be updated to show the new location of this page. The process of releasing a Smalltalk consists merely of flushing core to the current file, writing out the PRT, and closing the current file to further writes. From that time on, many people can simultaneously run that version by copying that page map and opening their own current file. Versioning serves both as a means of releasing systems to other users and of personal checkpoints. At any time a person can resume from the state of an earlier release operation. The cost is essentially the duplication of data dirtied between such releases. Automatic checkpointing (known as safing in OOZE) will be provided by a ring of current files. A page in a file belonging to an outdated checkpoint is recycled if it has been superseded by a copy in a younger file.

The PRT is run coded and may contain up to a million pages. If a system is composed of too many layers of releases or pages or two layers are too heavily intermixed, the PRT will be gigantic. We would like the PRT to remain in core to speed faulting. When the PRT becomes too fragmented, we will recopy entire blocks of pages to the current file. Many runs in the most

fragmented part of the PRT will thus be replaced with a single run.

ABell, Deutsch, Alan, Bobrow, Krasner, Horn, Althoff, Merry, Tesler, Robson, Goldstein, Shoch,
Weyer, Kachler, Ingalls, Marshall.WBST, Gifford, Lampson, Masinter, McCall, AHenderson,
(message to) McJones, MBrown, M Kay, Baldwin, Strollo, Fikes, DaveSmith, Lansford

Inter-Office Memorandum

To	Smalltalk Interest	Date	November 29, 1979
From	Larry Tesler	Location	Palo Alto
Subject	Smalltalk Memory System Requirements	Organization	PARC/SSL

XEROX

Filed on: <Tesler>MemSysReq.press

Introduction

This memo summarizes the goals of the planned Smalltalk-78 memory system and requests feedback from interested parties on the requirements that such a memory system must meet. The memory system design group especially invites comments about the needs of applications programs that require sharing of information between workstations.

Smalltalk-78

Smalltalk-78 is a reimplementation of Smalltalk-76 with the following differences:

- (1) *No Ooze.* The Ooze (object-oriented zoned environment) virtual memory system has been omitted.
- (2) *Fast object access.* A reference to an object in main memory from another object in main memory is implemented by an index into a resident object table which in turn points at the block of storage occupied by the referenced object.
- (3) *Contiguously stored processes.* The contexts that comprise a process are stored in a contiguous block of storage managed as (or like) a hardware stack.

In other words, Smalltalk-78 is implemented in a conventional way as a stack-oriented byte code interpreter with dynamic allocation of resident objects. Implementations on the Dorado, Alto, and Notetaker are at various stages of near completion. All available memory can be used. Performance of resident programs is roughly twice as fast as in Smalltalk-76.

Currently, the class library of Smalltalk-78 is a large subset of that of Smalltalk-76. From now until the end of 1980, LRG and friends are rewriting the class library to make it more concise, more consistent, and more flexible.

Virtual Memory and Public Memory

We are currently designing a new memory management system for Smalltalk-78. Among the active collaborators have been: Dan Ingalls (chairperson), Dan Bobrow, Peter Deutsch, Ira Goldstein, Austin Henderson, Ted Kaehler, Alan Kay, Glenn Krasner, Diana Merry, Richard Moore, Dave Robson, Larry Tesler, and Steve Weyer.

One goal of the system is to provide a *fast virtual memory*. The virtual memory should expand the address space well beyond the bounds of main memory. However, unlike Ooze, when all needed objects are resident, programs should run without virtual memory overhead. Working set changes should be accomplished with far less thrashing than in Ooze.

We wish to allow objects stored outside main memory to reside on both local disks and remote file servers. The virtual memory system must support the addressing of objects on all such external media, must support the transfer of objects from those media to main memory when needed, and must support the return of changed objects to those media under specified conditions.

A second goal of the system is to provide a *large public memory*, i.e., to allow sharing of information among workstations and servers.

A third goal is to allow detachment of a workstation (especially a Notetaker) from the network with a piece of the private and/or public memory saved locally, and to later reconnect to the network and reconcile any differences that have arisen between the local and remote environment.

It may or may not be true that the virtual memory system must support the public memory system and/or the detachment capability. An alternative would be to support sharing at a higher level. In either case, there may be advantages to making the swapping unit (page, segment, or whatever) be the same for both local swapping (disk $\leftrightarrow$ main memory) and remote swapping (public or private remote disk $\leftrightarrow$ local disk or main memory).

Applications of Private Memory

Reasons for having a private memory larger than the size of main memory are well known. A reason for allowing parts of that memory to exist on remote file servers is that Notetaker-like machines have disks unsuitable for fast swapping of large amounts of information.

Applications of Public Memory

There are several possible applications for public memory, including:

- * System release control
- * Program library sharing
- * Collaborative document preparation
- * Mail systems
- * Data base management
- * Distributed multiprocessing
- * Multiuser interaction with views of the same changing model

Implementation Characteristics of a Public Memory

Each type of application makes different demands on a public memory system. Below are listed a number of dimensions along which to characterize a public memory system:

- * Grain of transfer (field, object, page, segment, working set, message, user action, expression, relation, ...)
 - * Capacity (in units) of main memory & of local disk
 - * How much of a transferred unit is actually utilized
 - * Speed of transfer to/from public memory
- * Transaction processing host (local or remote)
- * Speed of communication between users (bandwidth and delay)
- * Consistency tradeoff (consistency with earlier access by this host vs. consistency with other hosts that are accessing the same information)
 - * Copies vs. originals
 - * Versions (PIE layers)
 - * Update delays
 - * Detached workstations
- * Crash recovery
- * Garbage collection
- * Range of host types (Notetaker, Dorado, etc.)
- * Affect of performance characteristics on programming style

An Issue

There is one hotly contested issue at this time, namely, how to provide for a wide range of delays in returning information to the public memory. For example, to run TeleSil-like applications, it is necessary to return information to the shared memory as soon as it changes. On the other hand, to share program libraries, it is desirable to defer the return of a changed version to the public memory until it is completed and debugged (as long as there is then a way to reconcile changes to the same programs by multiple users). Similarly, after a Notetaker has been detached from the network, the information in it can not be returned to the public memory until it is reconnected.

One theory holds that all applications in the fast-return category are, like TeleSil, implementable by sending short model-update messages directly to the other machines involved and bypassing the file servers entirely. Therefore, memory system designs may assume that changed information need not be returned rapidly to the public memory except in this special way.

The opposing theory holds that some interesting data base management systems require up-to-date views of overlapping information by multiple workstations. Therefore, memory system designs must have a way to return information rapidly to the public memory.

Request for Information

If you have implemented, hope to implement, used, or hope to use a system with shared memory, i.e., if you have thought about the requirements of one or more applications of public memory, please let the design group know (through me):

- * What is the application?
- * What is the user's perception of sharing?
- * How rapidly must views of public data on multiple workstations be reconciled when changes occur? How soon must changed data be returned to the public memory? Can all cases where fast return is required be handled by short messages to other workstations having active connections to this one?
- * What grains of transfer might be suitable?
- * Where does the application fall along the consistency tradeoff?
- * Is it important to implement the application on a portable machine like Notetaker? Is detachability important?
- * What crash recovery requirements are there, and what special reliability problems does the application pose?
- * How is garbage produced and how might it be detected?
- * How helpful or unhelpful would be each of the following representations of the shared data:
 - * B-tree
 - * Relational data base
 - * Individual objects
 - * Tree-structured independent objects
 - * Graph-structured independent objects
 - * Interrelated objects
 - * Paged linear address space
 - * Zoned versions of any of the above
 - * Multiple host versions of any of the above
- * Is there a natural naming system for the objects in the public memory?
- * In what ways might the virtual memory system profitably support access to public memory?

Any other helpful comments are invited.

Prompt replies are encouraged, even if they address only some of the issues.

Inter-Office Memorandum
DRAFT - DRAFT - Fairbairn, Ingalls, Goldberg, Jeffers

To	All Notetaker Buyers	Date	January 3, 1980
From	Larry Tesler	Location	Palo Alto
Subject	How Much Memory for Your Notetaker?	Organization	PARC/SSL

XEROX

Filed on: <Tesler>NTMemAmt.memo

Introduction

A Notetaker II can have anywhere from 128 KB (131,072 bytes) to 1 MB (1,048,576 bytes) of fast read/write memory, sharable among one or more processors. The smallest size Notetaker has the same amount of memory as a minimum Alto; the largest has twice as much as a maximum Alto. In addition, each processor board in the Notetaker has a few thousand bytes of its own very fast memory.

If you are buying a Notetaker, you must decide how much main memory to order. It may be useful to you to know how much memory LRG -- the Learning Research Group at PARC -- is ordering for our many Notetakers, and the reasons we are doing so. (LRG advised and assisted in the development of the Notetaker hardware, and is developing Smalltalk-78, its standard software system.)

Hardware Options

The attached memo from SPG explains the hardware options and their prices. An important fact to note is that if your machine is configured in the standard way (including three memory boards of 128 KB each, two general-purpose processor boards, and an Ethernet board), then all the card slots in the cabinet are used up, and the power supply is stretched to its limit.

Some users may need to insert additional boards for homebrew hardware, for more processors, or for more memory. There are three ways one can free up card slots: remove memory boards, remove one of the general-purpose processor boards, or remove the Ethernet board.

Before deciding to remove memory boards, note that to run Smalltalk-78 requires at least 256 KB -- and preferably 384 KB -- of main memory. Memory requirements are discussed further on the next page.

Before deciding to remove a processor board, note that Smalltalk runs a little slower and must pause frequently to service the disk and other I/O devices when it must handle all its tasks with only one general-purpose processor board.

Before deciding to remove the Ethernet board, note that fast communication with file servers, printers, Altos, and other Notetakers depends on it. Be sure your communication requirements can be met by an EIA interface.

Smalltalk Requirements

We expect most users of Notetaker to program in the Smalltalk-78 language. Smalltalk-78 programs can be run interchangeably on Notetaker and Alto (and also Dorado), as long as they don't depend on idiosyncratic I/O devices. The system includes an incremental compiler, an automatic storage management system, a simple text and program editor, and a substantial library of tools for program development, calculation, text manipulation, simulation, and graphics.

Smalltalk's extensive capabilities require a lot of memory space. Today's version of Smalltalk-78 occupies 231 KB of main memory, including a full-screen bit map, leaving 25 KB for user programs. This number will change between now and the system release planned for later this year.

Dan Ingalls and I estimate that the space available for user programs in a 256 KB machine will be between 16 KB and 48 KB, enough for between 20 and 100 pages of Smalltalk source code, less whatever space is required for data. In a 384 KB machine, we estimate that there will be between 144 KB and 176 KB available for user programs, enough for between 300 and 400 pages of Smalltalk, less data space. Thus, the larger memory size will accommodate programs at least four times as large -- and possibly fifteen times as large -- as will the smaller size, for an extra cost of a few hundred dollars.

It is possible to develop programs in Smalltalk and then to generate an execution-only version that takes substantially less space. One use of this capability is to develop application programs for a small Notetaker using a larger development machine, which could be an Alto, a Dorado, or a Notetaker with extra memory. LRG expects to configure some of our many Notetakers with 512 KB of memory for development work and some with 128 KB for execution-only use, but we expect most of our machines to have 384 KB.

LRG is working on a virtual memory system for Notetaker Smalltalk-78 that will use a network file server as a swapping store. After the virtual memory system is finished and released, internal memory will be less of a constraint. But remember, one of the major reasons for using Notetaker is to be able to detach it from the network and carry it to distant sites. Even after virtual memory is available, 384 KB or more of internal memory will still have an advantage for the many applications that require portability.

Recommendation

It costs only a few hundred dollars more to have 384 KB of main memory than it does to have 256 KB. It is a real bargain. If you don't need spare card slots, buy 384 KB. You will be able to run much larger programs.

If you do need a spare card slot, order 384 KB anyway, and first try removing a processor board. If the performance degrades too much, try removing memory instead. If the programs you plan for the Notetaker are 20 pages or less of Smalltalk, and don't require much data space, then you may be able to get by with 256 KB. If you are planning to obtain software from another source, such as Intel, and do not need Smalltalk at all, then you may be able to get by with 128 KB or 256 KB.

If you need more than 384 KB, you can buy an extra memory board and remove a processor board, yielding 512 KB.

If you need 768 KB or 1 MB, or if you need several spare card slots, contact Doug Fairbairn or Doug Stewart about the possibility of obtaining memory boards stuffed with 64 kilobit chips instead of 32 kilobit chips. They should be available at a somewhat high price late this year.

If you have further questions, do not hesitate to contact me. My Intelnet number is 8-923-4355; from the local calling area: (415) 494-4355; electronic mailbox: Tesler.PA; mail stn PARC 35-2033.

XEROX

PALO ALTO RESEARCH CENTER

Learning Research Group

July 27, 1976

To: Virtual Memory Interest

From: Dan Ingalls, Ted Kaehler

Subject: The OOZE Virtual Memory

Filed on: <INGALLS>OOZE.Bravo

Dan Ingalls
2/10/77

Abstract: This paper describes the OOZE virtual memory. Its chief features include:

large address space - 2^n objects addressed with n bits

dense workingset - only the active objects occupy core

reduced disk activity - swapout is by zone

further savings - structural match with SMALLTALK

The description presumes no expertise in virtual memory, but it may be confusing if you are not acquainted with such concepts as 16-bit word, core address, buffer, hash table, etc.

I. Introduction to OOZE

OOZE (*object-oriented zoned environment*) is a virtual memory system. This means that it will translate from a virtual address to a core address where the data is held, and if not in core, then it will arrange to make room for the data and bring it into core as well. OOZE's key features can be illustrated by contrast with another well known virtual memory technique called *paging*. For the purposes of discussion, we will assume that 16-bit words are used.

A typical 16-bit paging system would be able to address 65536 words, and addresses would be treated as two 8-bit fields called the *page* and the *offset*. What real memory is available would be treated as a pool of 256-word buffers for holding pages. When a program accesses a given address, the *page* is looked up in a *page map* to find its base address in core, and the offset is then added, yielding the final memory address. If the page is not in core, the *page map* reveals where it is on the disk, as well as what page should be thrown out to make room for the new one.

A 16-bit OOZE would be able to address 65536 *objects*, and the address space is divided into equal *zones*. Each *zone* contains *items* of equal size, and an object pointer may be treated as two fields indicating the *zone* and *item-number*. When a program accesses a word, it provides both an object pointer and a word offset relative to the object, each 16 bits. The object pointer is looked up in a *resident object table (ROT)* to find its base address in core, and the offset is then added to get the final memory address. If the object is not in core, then the disk address is computed from a *zone index of pages*, and a place in core is allocated for the object. If available memory runs out, then a sweep is made of the ROT, to swap out objects which have not been recently referenced.

Figure 1 illustrates the structure of OOZE. Most interesting aspects of OOZE will arise by pondering this structure in the light of several performance criteria.

Access to resident data. While a page map is a simply indexed table, OOZE's *ROT* is accessed by hashing, and consequently the indirection is slower. The delay can be minimized by microcoding the ROT access, and it can be eliminated through the use of base registers.

Base registers. If data is accessed through base registers, the *hashing need only be done once* to load the base register, after which the access time is the sum of the register access, the offset add, and the final data reference. The improvement actually realized with base registers is obviously related to the mean number of accesses made while a register remains pointing at one object. Paging can also be assisted with base registers, but the gain in speed is not so great unless the paging space is large enough to require a hashed page map.

Space utilization. One advantage of OOZE is that it holds in memory *only the objects which it really needs*. If only 3 words are needed, then 3 words are used, whereas the paging system is forced to bring in a full 256 words, or whatever the page size is. The extent to which this is an advantage can be seen by looking at the mean number of words referenced in a paging system during the time that page is in core (and this depends strongly on the way in which storage is allocated). The ROT space counteracts this advantage somewhat, since it carries one mapping entry per object, rather than one per page. The representation of immediate data, and the mean resident object size are important to the analysis of space utilization.

Immediate data. The general structure of OOZE is similar to other segmented virtual memories. The central difference is the grain size - OOZE handles every datum, down to single word integers, in the system which it supports. The ROT consists of one 2-word entry for each object in core, but one exception reduces the overhead considerably. If an object has only one data word (integer, 1- or 2-character string or atom) then that *data word is placed directly in the ROT* where the core address would have been. This reduces the overhead to one word in this important case (see also *atoms* and *small integers* below).

Mean object size. A goal of OOZE is to provide a large address space without going to pointers larger than 16 bits. This is certainly achieved if the mean object size is on the order of 10 words, since the effective address space is greater than 500K words. At the same time, such a statistic seems to belie our claim of significant savings from immediate data, since there must not be many small objects. *This is not a contradiction*, on the one hand we are looking at the whole address space statically, and on the other hand we are looking at the statistics of the workingset dynamically.

Changing context. It is more difficult for OOZE to change from one context to another than it is for a simple paging system, since each new object has to be brought in separately. In paging, many of the new objects get brought in fortuitously because they lie on the same page as other objects being referenced. OOZE includes a pool of disk buffers so that it too can enjoy this "neighborhood" phenomenon, especially in the case that the faults refer to objects of the same zone.

Swapping. Possibly the greatest obstacle to implementing an object-oriented memory at the grain of single words is deciding what to throw out when new data has to be brought in to the workingset. A scheme which works for large objects is to throw out the first thing you find which has not been referenced since the last swapout. At small grain, though, it may take more than 50 scattered writes to the disk before enough room becomes available. OOZE's solution is to *swap on the basis of zones*. Whenever space is needed, a new zone is purged, meaning that all resident objects in that zone which are old get swapped out. To this end, each entry in the ROT carries an older-than-purge (OTP) bit which tells if it has not been referenced since the last time its zone was purged. The

cup zones on the
ouch, since the

zone purge solves the disk scatter problem, since some effort is made to group zones on the disk. It also aids enormously in recording how recently data has been touched, since the OTP bit has a *much longer history* than just since the last swap.

Resident data management. A certain amount of complexity is involved with handling a purge. The ROT entries have to be deleted, and the resident data area has to be compacted, with allowance made for objects which have been locked to fix their core address.

Contiguity of resident data. *Resident objects are stored contiguously*, so that no pain is caused by objects which would straddle page boundaries in a paging system. There is, however, an effective upper limit on the size of objects which can be handled, since a few large objects could exhaust the available resident data space. *OOZE provides for locking* of resident objects to guarantee their core addresses, especially across purges. In this way, time-critical system functions can bypass the ROT mapping entirely.

OOZE was conceived to serve the needs of a high-level programming system, SMALLTALK. Much of this service would be the same for APL, or LISP, and we cover some of the considerations here as an introduction to their treatment in the next section.

Classes and zones. A SMALLTALK class is like a data type, and its instances are typically all the same length. It therefore makes sense to allocate the instances all within a zone. If a class has more instances, then it can have several constituent zones. A table, called the zone map, reveals the mapping from zone number to class, and thus allows the class of an object to be computed indirectly from its object pointer. In this way, there is *no need for an additional class pointer* (a 1-word per object overhead), and yet flexible allocation of the address space among classes (in blocks of 128) is achieved. The zone map also carries a field for instance size, so that the *blocks of resident data do not each need a size field*, and it carries other information such as whether the object is a pointer-type object or not.

Variable length data. Classes with variable length instances, notably vectors (linear arrays of 16-bit object pointers) and strings (linear arrays of 8-bit data bytes), have many zones for each of the possible lengths. Thus the length of a 4-character string is revealed by the zone part of its object pointer, and thus needn't be stored in the resident data, another 1-word saving. Note especially that 1- and 2- character strings (and atoms) can fit in only 1 word, and thus qualify further for the immediate data savings!

The allocation of a zone (or more) per length would be wasteful for all but the most common lengths, so for lengths greater than 7, the zones cover octaves of data size (*ie.* 8-15, 16-31, etc.). On the disk, a 23-word variable object would waste 7 words but, when brought into core, it would occupy 23 words preceded by a length word.

Reference counting. OOZE supports reference counting by providing 4 bits in the ROT for a reference count. Counts which exceed 15 are handled in a separate hash table, and objects on the disk have a full word for their reference counts. When freeing an object, its entry in the zone map reveals whether it is a pointer-type (most SMALLTALK objects are), in which case the reference counts of all the objects pointed to by its subfields are decremented also.

Representation of atoms. Atoms are strings used for names, and they have 4 other properties in this system:

- 1) They are not reference counted
- 2) Their object pointer is a unique function of their characters
- 3) They cannot be written into, once created
- 4) Only explicit reference to their characters requires residence

The OOZE address space is allocated with atoms growing down from the top, and everything else growing up from the bottom. In this way, a pointer can be discovered to

be an atom simply by comparison with the minimum atom pointer so far allocated.

Small integers. Integers in the range of -1000 to +1000 are also allocated at the top, above the atoms, and these, too, avoid the overhead of reference counting. Moreover, since the object pointer reveals the value, these small integers require *no resident data space nor even an entry in the ROT!*

Checkpointing. In order to recover from disasters, our scheme uses a simple mechanism for checkpointing. Every so often, the entire core image of the system is written as a header at the beginning of the OOZE virtual memory file. Thereafter, until every zone has been purged again, SMALLTALK can be restarted by resuming that saved image. The reason is that everything which can get smashed on the disk by subsequent purges exists as a clean copy in the saved image. Only after a second round of purges starts does the checkpoint become invalid.

Inter-Office Memorandum

To OOZE followers

Date

June 9, 1975

From Dan Ingalls

Location

Palo Alto

Subject Evolution of OOZE

Organization LRG

XEROX

INTRODUCTION

Since the last memo, I have changed my view of OOZE, and the current view is outlined below (still a very rough draft). The transition followed roughly the following path:

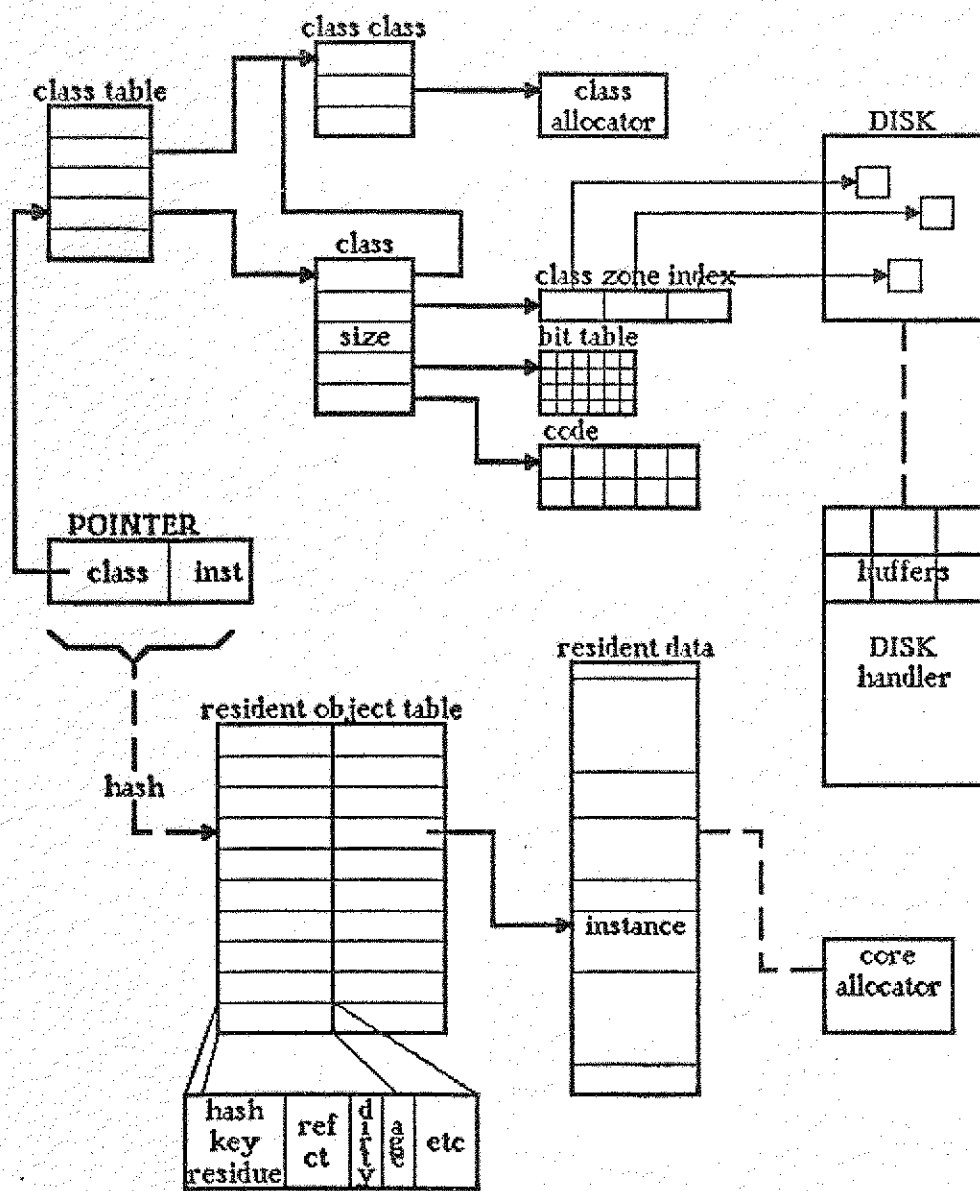
1. Butler said he was unimpressed by the idea of having resident and virtual data representations be identical, especially since any conversion between the two would be lost in the disk access hiccup.
2. With that in mind, it seemed to make sense to zone the resident data by class, to allow a more compact representation in core.
3. At that point it seemed silly not to have the virtual format also be zoned, and so I revived the pointer format which I had lightly considered last fall, and which is described below.
4. The in-core zoning looks ugly to manage, so resident data is not zoned in the sense of being located together, but the new format does reveal the class of an object, so in that sense the factoring of class information is still achieved.
5. The zoning leads to a very nice (I think) solution to a big problem of object-oriented virtual memory - namely a strategy for discarding old objects. See below.

As you may gather, OOZE has become an integral part of SMALLTALK rather than some general purpose data mapping utility. Several important issues remain to be worked out. Most important is management of the resident storage space.

POINTER FORMAT

A pointer is a 16-bit quantity, split into a 10-bit class number and a 6-bit instance number. This design allows up to 1024 classes, and at least 64 pointers are allocated for each class. The class number indexes a class table to get the actual class pointer. If a class grows to include more than 64 instances, another class number can be made to refer to that same class, thus extending its portion of the address space.

OOZE is thus capable of addressing roughly 60K objects. If the mean object size is 10 words, this is half of a model 31 disk, or 10 times the space available in the current paging SMALLTALK.



ALLOCATION OF INSTANCES

Each class holds information relevant to the allocation of its instances. For instance, a class of x-y coordinate pairs would have a field for its instance size (=2 words), a field which points to a list of disk pages which hold the instances of this class, and a field which points to a bittable for managing the free space in this class zone of the disk.

Naturally, some classes will want to be allocated in a different manner from others, for instance vectors which are not fixed length. What this means is that those classes are instances of a different class class. It is really the class class which points to the specific allocator to be used by its members. This organization already obtains in spirit in Dave's implementation of SMALLTALK.

MAPPING

Obviously, the indirection through class number represents a critical delay, and asking the class to map the instance number (seems the cleanest conceptually) would clearly fail in performance. Therefore a resident object table (ROT) is provided to associate a base core address with any pointer whose instance is in core.

An entry in the ROT includes 16 bits of core pointer which locates the raw data of the designated instance. It also provides 6 bits of hash key residue, 1 bit of cleanliness, 1 bit of age (godliness), 4 bits of reference count (with escape to an overflow table), and sundry other flags as needed.

Depending on statistics, we may use one of the flags to indicate that the instance is a one-word datum, and is located immediately in place of the core address.

The mapping through this table will be done with microcode and some base registers.

SWAPPING

Object-oriented swapping is a potential hazard to performance. When the workingset changes appreciably, there may be 300 new instances to bring in, rather than 10 new pages. To avoid thrashing in this situation, the disk handler will have many buffers to window the data on the disk. Hopefully ten or less buffers will suffice, lest all our core be wasted trying to achieve the behavior of paging.

Swapping out is also a problem, as it is not of much value to throw any one instance out of core. Instead, many must be thrown out, and this leads to a dilemma in how to choose the many (heavy aging problem), and an even worse problem about scattering disk writes, since rejects surely don't group as well as demands (though there are typically fewer of them).

Classes to the rescue! Purges of the resident data are done on a class by class basis. That is, a sweep of the ROT is made, and all instances of the currently purging class which have not been touched since the last purge of this class (that's what the age bit means) are written out. This seems like a nice strategy for partitioning the resident data to swap out. It has the wonderful feature that the disk access will be localized because instances of a class are zoned to a group of disk pages.

MEMORY MANAGEMENT

Some further thought has to go into the management of the resident data area. It's not hard to think up appropriate reservation and liberation algorithms, but the problem here is compaction, since the ROT must be fixed up to do a compaction. It would be nice to do this incrementally, natch.

The design of OOZE ensures that all instances are mapped into contiguous core, thus solving half the problem of virtual code segments and other similar needs such as fonts and music timbres. The other half of the problem is to allow for locking these structures into core, preferably with provision for moving them. This is another detail that remains to be worked out.

IMPLEMENTATION

I hope to finish the design of OOZE by the end of June, and do the implementation during July. Toward the end of July, there will be a Frankenstein party where legs of OOZE will be grafted onto the body of Dave's SMALLTALK. This operation will take longer than we estimate, but when we're done we will know whether OOZE is an existence proof, or just an experiment.

P
D

CH 12.

Outline of Virtual Memory

0. Packing the U-Haul trailer

1. Why a virtual memory? What does it do?
2. Data flow maps, how to read them.
3. The Schemes:

Smalltalk 72

Paging Smalltalk

OOZE (Smalltalk 76 and Fastalk)

NoteTaker

4. What we have discovered!!

Standard interfaces
Collapse levels for speed
famous tricks

5. Large virtual memories (findit vs. ooze)

Designing Virtual Memories --- What I Learned From Packing a U-Haul Trailer.

Everyone would like both more space and speed in the new implementation of their favorite computer language. The limits of the hardware are usually etched in silicon before the design ever starts, so the problem is to do the best with what you have. You start out with this empty trailer (the machine) which you know is too small. You have a pile of boxes (services the vmem is suppose to provide; some of them must go fast), which are suppose to fit in the trailer. You can leave some of them out if you get in a pinch, but the people who are attached to them will be very unhappy. Success is when all the boxes fit into the trailer and the door closes.

Form follows function in virtual memory design. There may be several different solutions to the packing problem and the different solutions are not usually near to each other. (To get from one good solution to another, you usually have to take everything out and move the biggest box!!) Just as in packing, each decision you make influences the rest of the design. There are things you can do to control these complicated interactions. You can decide to fill the trailer up to a flat level, then the problem of filling the second half is separate from the first half. (Each is simpler than the whole thing). This is called layering. The standard practice of separating the paging virtual memory from the operating system and programming language turns the vmem into a separate layer.

The problem with layers is that they cost time (and space). (Many computers use special paging hardware to eliminate this time). The trick is to mix up several layers in order to take shortcuts without compromising the function of the top layer. If there is a way in which the boxes fit perfectly into the trailer, then you win and everything works. If no matter how you juggle things, there are three small spaces and one big box left over at the end, then layering might have been a better idea.

This chapter tells about four different ways we loaded the trailer and what we learned from it.

Virtual Memory

Smalltalk is an object oriented language. The fields of objects contain references (pointers) to other objects. New objects may be created. Each object is a member of some class. Each object has a certain number of fields. The fields of an object are either pointers to other objects or just raw bits (whose meaning is known only to the object's class). No explicit mention of the death of an object is made in

KARLA, COULD YOU MAKE A COUPLE
COPIES OF ALL OF THIS AND ASK
DAN AND DAVE TO COMMENT ON
THE FIGURES?

24

du -
Storage allocation on
demand dynamic
supports always
garbage collection

unafraid

a different use of the
what? it is close but we have to work on this
fun!!

the definition of Smalltalk. However, because of the finite bounds of space, objects that are never going to be used again must be recycled in some manner.

In response to a request to create a new object of a certain class, the virtual memory must provide space for the number of fields in instances of that class. Pointers to other objects stored in the fields must be preserved, so that those objects can actually be referenced on demand. The pointer to the class of a given object must be produced on demand. The number of fields in an object (especially a variable length object) must be provided upon demand. Objects must be destroyed at the proper time (sometime after the object can never be referenced again. Exactly when is a matter of optimization).

All this must be done on a particular machine with a particular architecture and primitive instructions. The design of a virtual memory is a sensitive function of the exact machine design, the size and speed anticipated for the finished Smalltalk, and various specific details of what is suppose to go fast or allowed to be big in the Smalltalk. The four virtual memory systems detailed below are a wide spectrum of designs.

Data Path Charts

The conversion of a collection of bits to an address in some address space is so automatic in a computer that we normally don't think about it. Since much of a virtual memory design depends on the details of the the conversion of bits into addresses, it is important to show it explicitly. Figure 12.1 shows the conversion of a 16 bit word into a one-in-64K address. The rectangle at the top is the 16 bit word. The square with the diagonal line signifies the conversion lot the point of view of an address. Figure 12.2 shows a 15 bit quantity divided into two fields (6 and 9 bits wide). This quantity addresses the even values of a 64K space because a zero is appended at the right. (Low order bits are at the right, high order bits are at the left). On the address side, the space is divided into a 'heap' from 0 to A (A is some variable), a free space from A to B and a 'stack' from B to 64K. The arrows signify that the stack grows down and the heap grows up. The words *every 2* in the stack area mean that some sort of boundary exists on even values. The words *some 2* in the heap mean that a boundary exists on some even value addresses.

Figure 12.3 shows an 8 bit value corresponding to a 0 to 255 address which indexes a piece of physical memory. The address is a word address and the memory indexed starts at a location called 'table' in physical memory. (The 8 bit number is an index off the value 'table'). The result of a lookup in the memory is shown at the left. The whole word is called 'ABC'. It is subdivided into fields E, F, and G which are 8, 4, and 4 bits wide.

The diagrams show data flow only, and not algorithms. The paths do not have directionality. The address to bits correspondence may be traversed either way. The exception is the impossibility of deducing the memory address from the contents of the memory location. Braces, lines, and pairs of dotted lines show equivalence of different parts of the diagram.

The four designs

FIG 12.1

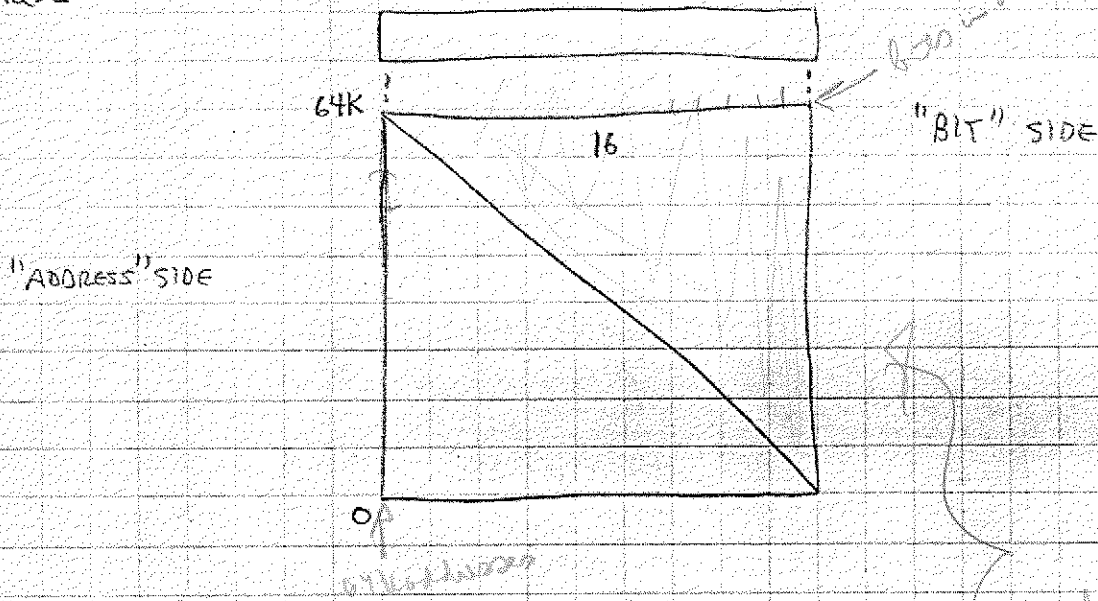


FIG 12.2

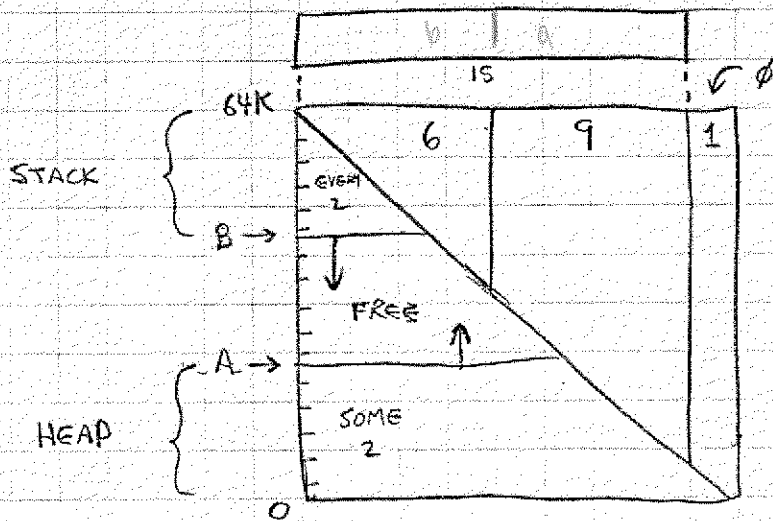
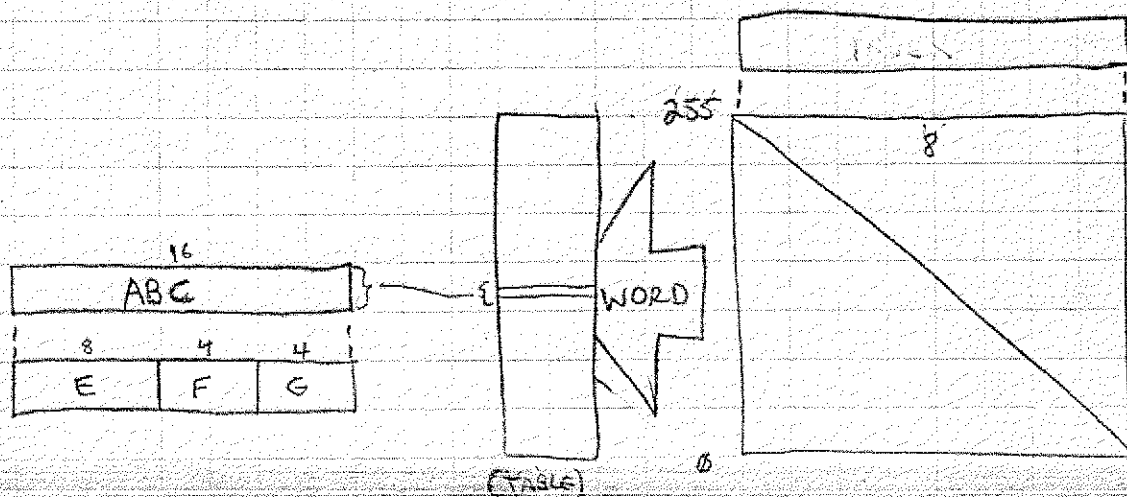


FIG 12.3



Physical memory is replacing
memory addresses
mapping for well # three
grounding & memory of
memory. How can we
achieve learning as in a day?

The virtual memory behind Smalltalk 76 is the Object Oriented Zoned Environment (OOZE). The design of ooze was driven by a set of constraints. We tried to find the simplest implementation that met all the constraints.

1. Ooze must provide for the creation of objects of various classes, storing of their fields, recycling of the object when it is dead, finding the class and length of an object (this is as usual).
2. The machine is the Alto with 16 bit words and 64K of memory. The disk is a Diablo model 31 removable pack with rotation time of 40 ms (give this in microcycles? or in byte code times?), an average seek time of 70 ms, and 4872×256 words capacity.
3. We wanted to address 64K objects. Since the average object size is about 8 words, this fit well with $20 \times 64K$ words on the disk.
4. Core devoted to object currently in use must be packed. Every word in the user core must be a member of an object needed in the computation.
5. A maximum of one disk seek per fault on an object is allowed.

One way to satisfy constraint 5 is to keep a table in core of the disk location of each object. This table however, would be considerably larger than core. Thus we were forced to organize the disk. If objects of the same length are placed together on the disk, and if those objects have consecutive object numbers, then their disk address may be calculated from their object number. Note also that not only the disk location of the object is encoded in the object number, but also the length of the object. Since all instances of a class have the same length, why not also encode the class into the object number by only permitting instances on one class in a single zone on the disk.

The object number is called an OOP (Ooze Object Pointer). The oop is divided into two fields, the Pclass and the Pinst. A pclass consists of 128 consecutive oops. All objects in a pclass are members of the same class (a class may own as many pclasses as it needs). All members of a pclass are the same length and reside in the same zone on the disk. Figure 12.4a shows how the oop is decoded into a Pclass Map (PM) entry and how it encodes the class. Figure 12.4d shows how the PM entry and the Pinst determine the disk address. Note that classes are objects like any other, but that they are segregated to the low oops.

By constraint 4 objects are packed in core. The ROT or Resident Object Table holds the current core address of the object. The ROT is a hash table with multiple reprobes. Since finding a field of an object given its oop is a high bandwidth operation, the hash function is in microcode. Figure 12.4b shows the path from an oop to the fields of the object in core. Note that object with only one word of data are stored in the word in the ROT that is usually the core address.

Some classes have variable lengths of instances. We allow this by giving the class many pclasses of different lengths. The common short lengths 0 to 8 each have a pclass of their own. The longer lengths

are covered by octaves. A single pclass (of length 16 on the disk) covers lengths 9 to 16. Likewise there is a pclass for each power of two of length. Members of octave pclasses have a length word for each instance to tell its true length. (When the object is in core, it occupies only as much space as it needs). See Figure 12.4c, core formats of objects.

Compaction, purging, aging, some obj not refct, why fault of object just to change its refct.

Additional constraints:

6. Checkpointing.
7. Small integers.
8. Dirty Bits. (how interact with refct)
9. Locking.

the rest of the Figures:

- 12.4e Reference counts, The core allocator, locking of objects.
- 12.4f Physical core, Small integers.

NoteTaker Smalltalk (figures)

- 12.5a Physical core, oops, integers, normal objects
- 12.5b reference counts, class of an object (class table)
- 12.5c Formats of an Object
- 12.5d Format of a class.

What we learned

FIG 12.6

Design tradeoffs in virtual memory (discuss the chart showing this. currently a stack of note cards).

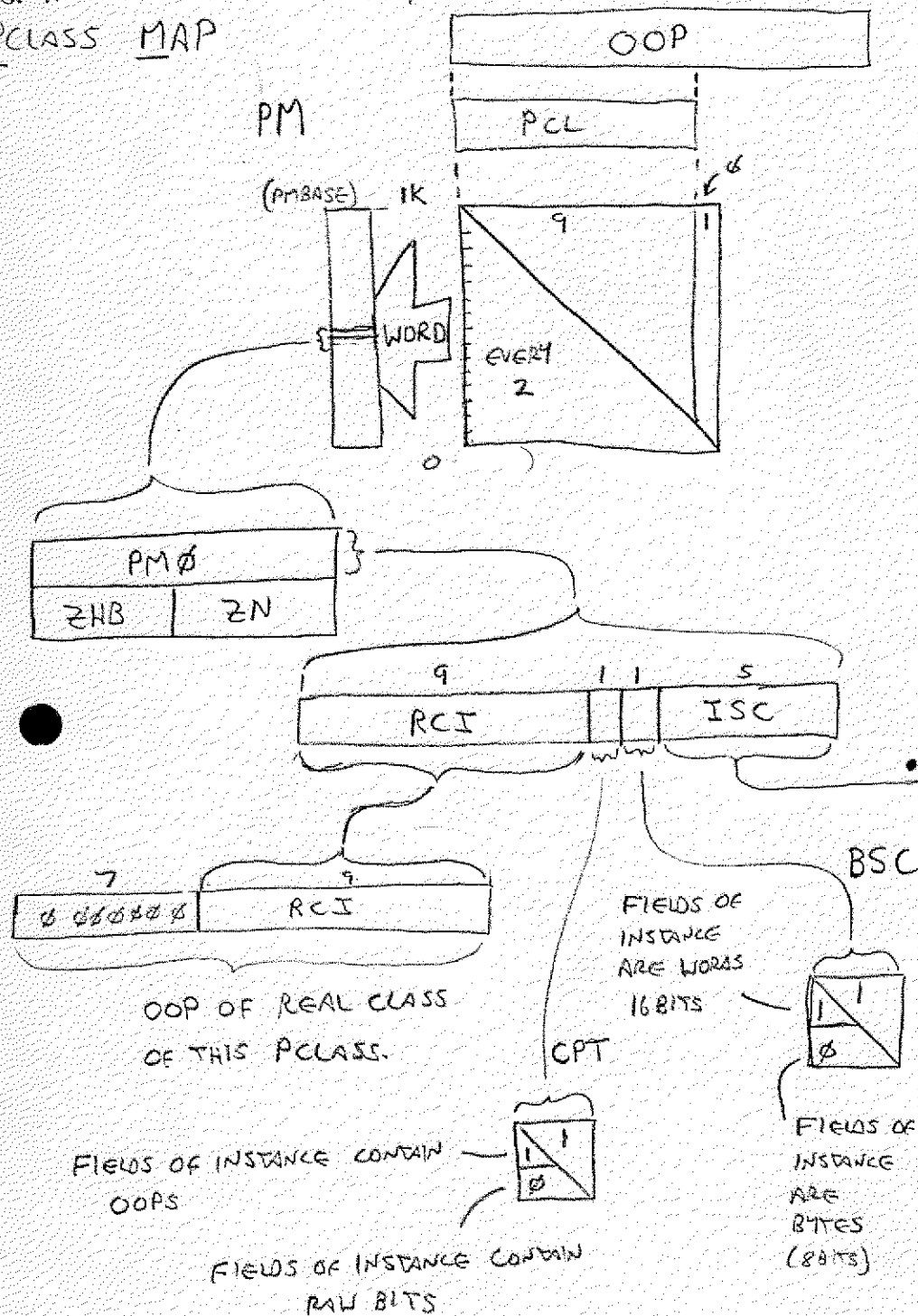
(Notes to myself:

1. draw pics for smalltalk 72, paging smalltalk.
2. compare the above with ooze memos from 1975.
3. dream -- three level dead object recovery. Name composting (stack peeling), ref count, global garbage collection
4. dream -- unified very large address space vmem (findit). shells of name spaces, each kept track of by one outside it.
5. draw a good picture of the design tradeoffs as we know them.
6. statistics -- back up certain decisions we made??

)

FIG. 2.4a

PCLASS MAP



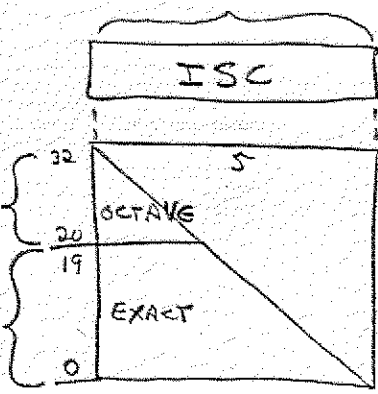
12.9.9

FIXED LENGTH CLASS

$$RCI < FIRVAROOP$$

LOOK IN INSTANCE LENGTH
FIELD FOR NUMBER
OF FIELDS

NUMBER OF
FIELDS IN
INSTANCE



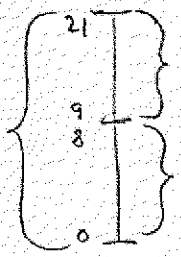
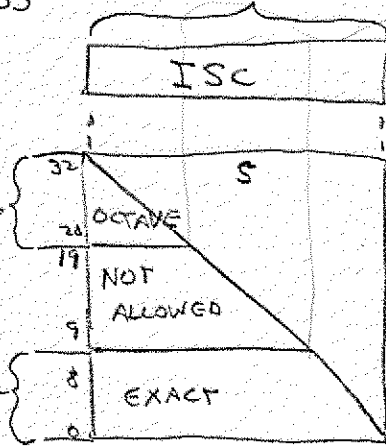
FREELIST
OFFSET
IN CLASS = ϕ

VARIABLE LENGTH CLASS

$$RCI \geq FIRVAROOP$$

LOOK IN LENGTH FIELD
OF INSTANCE FOR NUMBER
OF FIELDS

NUMBER OF FIELDS
IN INSTANCE



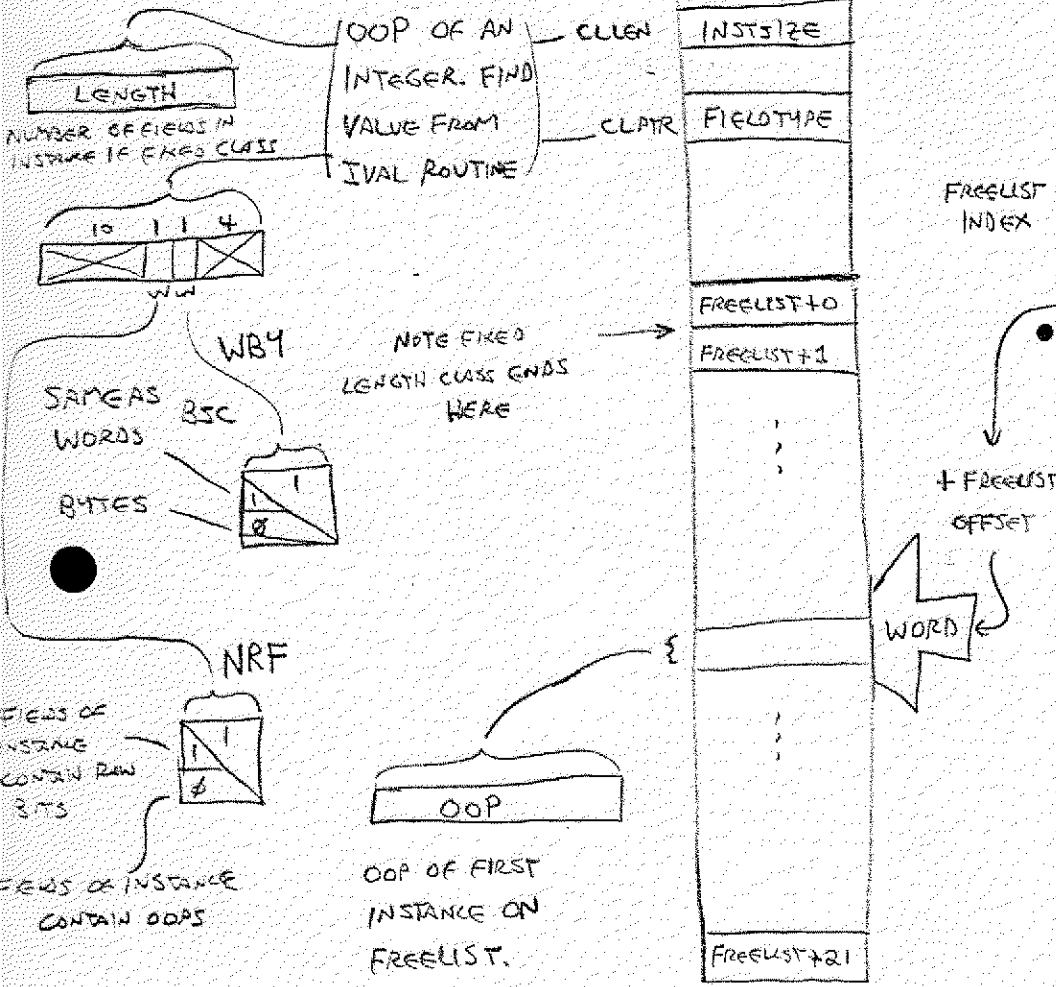
OFFSET OF FREELIST
IN CLASS

confusing

12.4a

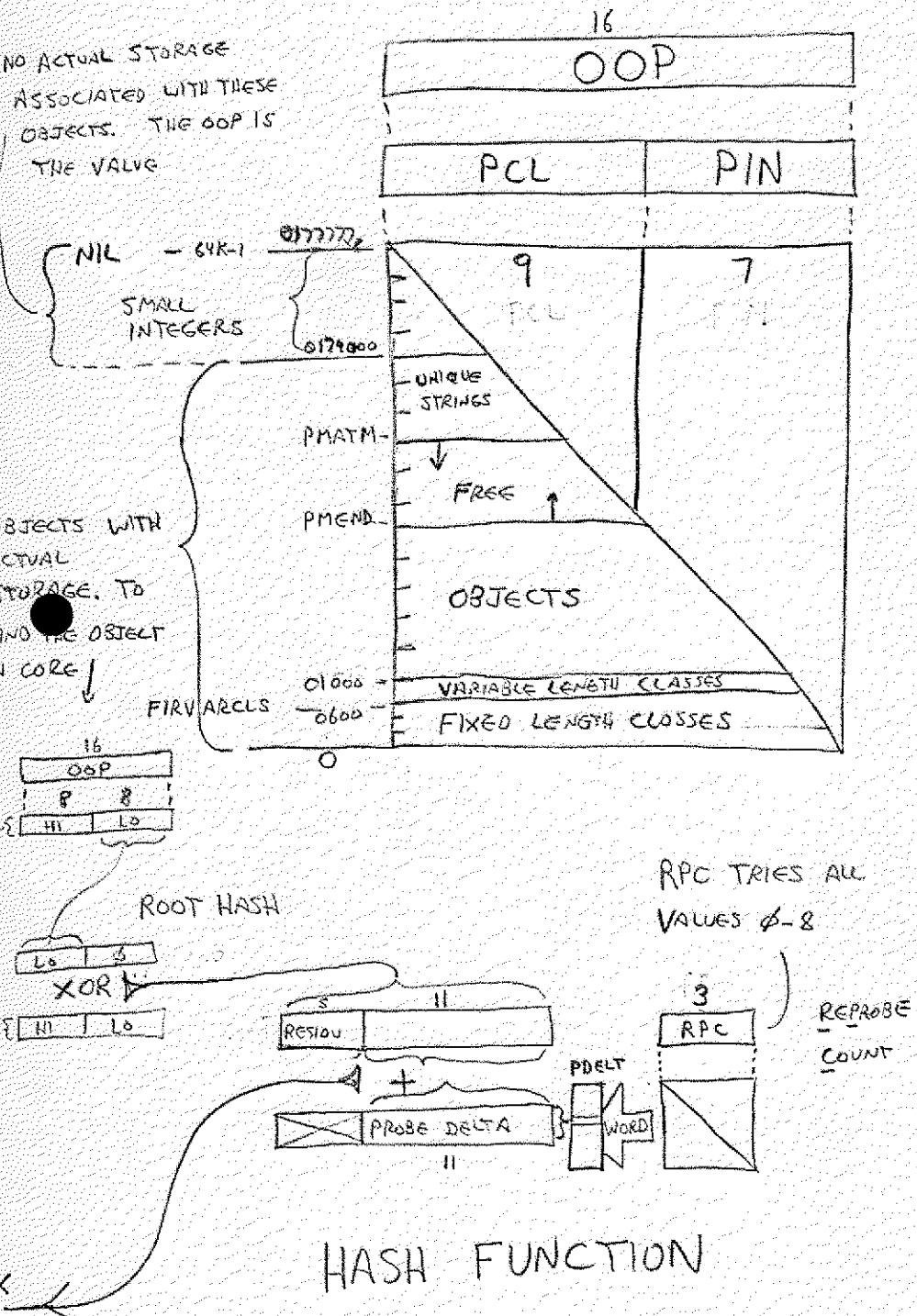
FORMAT OF A CLASS

(CORE ADDRESS OF CLASS)



SEE TABLE OF OBJECT
FORMATS FOR FREELIST FORMAT

FIG 12.4b



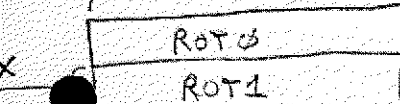
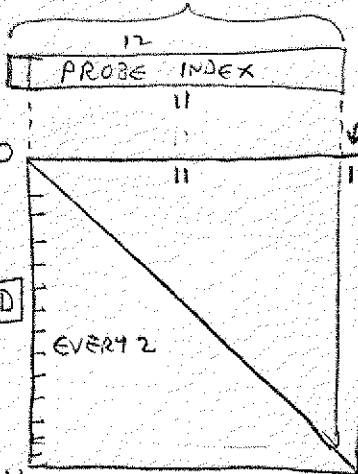
RESIDENT
OBJECT
TABLE

ROT

(ROTBASE)

WORD

4K



CORE ADDRESS OF
OBJECT IF
IMD = 0
ELSE FIELD 1



MUST MATCH FOR A "HIT"
RESIDU RPC

HASH KEY
RESIDUE

OTP



OBJECT HAS
BEEN TOUCHED
SINCE ITS PCASS
WAS PURGED

OBJECT HAS
NOT BEEN
TOUCHED SINCE
ITS PCASS
PURGED

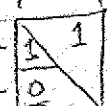
OLDER THAN
PURGE

IMD



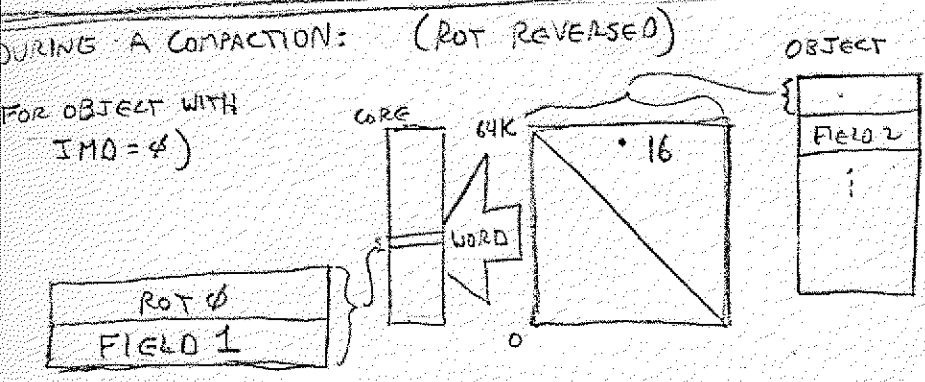
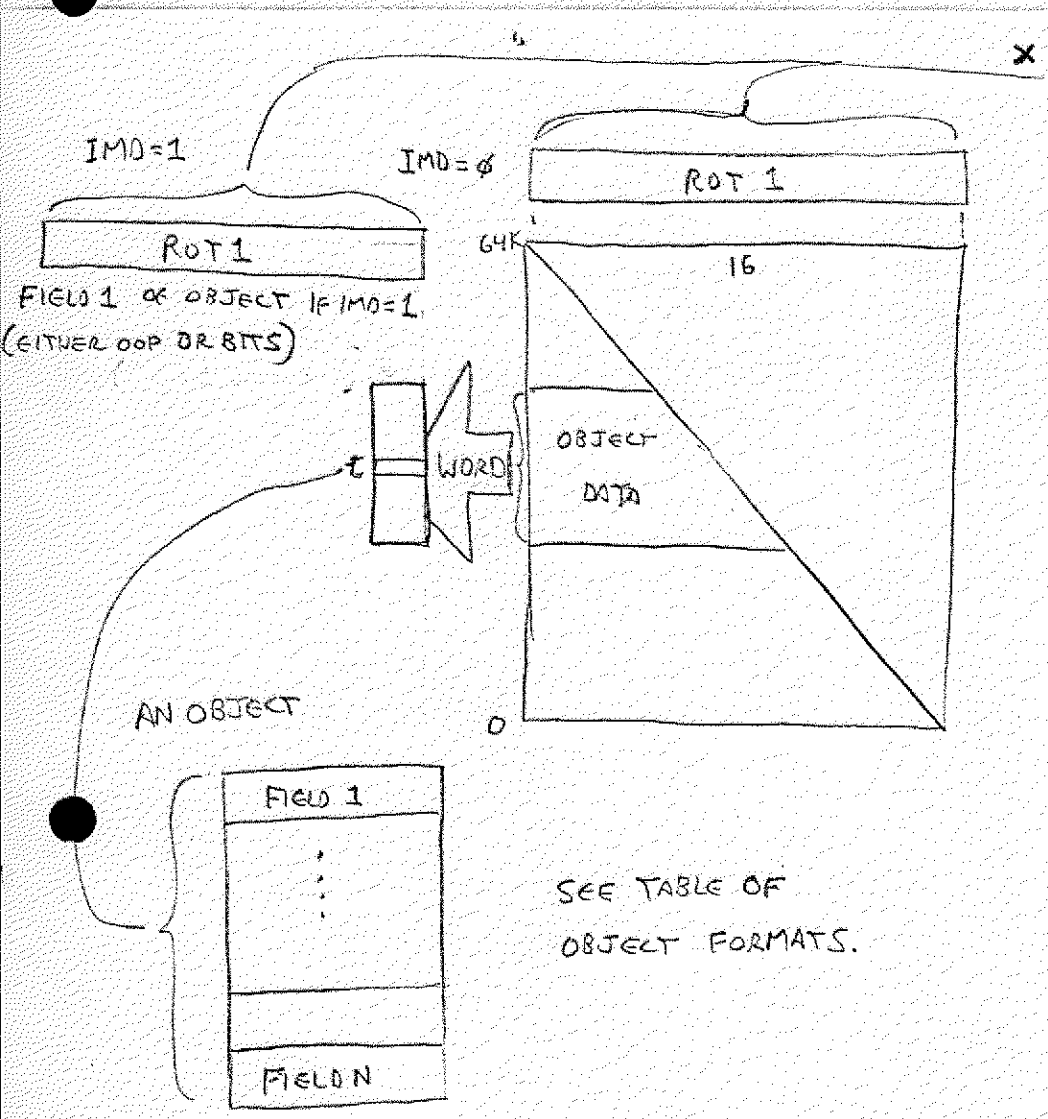
ROT 1 IS
CORE ADDRESS
OF OBJECT
ROT 1 IS FIELD 1
OF OBJECT

DTY

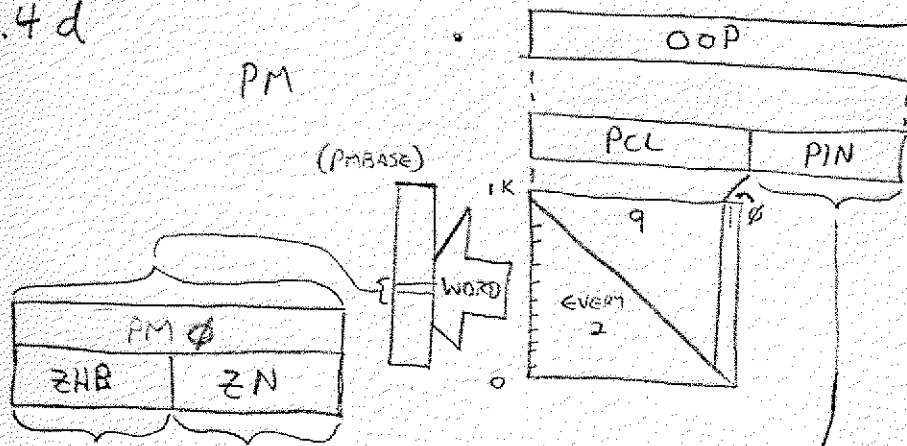


OBJECT IS
CLEAN
OBJECT IS
DIRTY

RCT



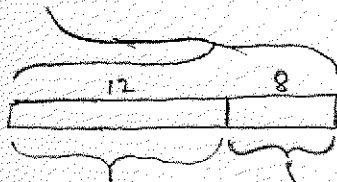
12.4 d



DISK LENGTH
DLEN

WORD IN ZONE

$$WIZ \leftarrow ZHB * 128 * DLEN + PIN * DLEN$$



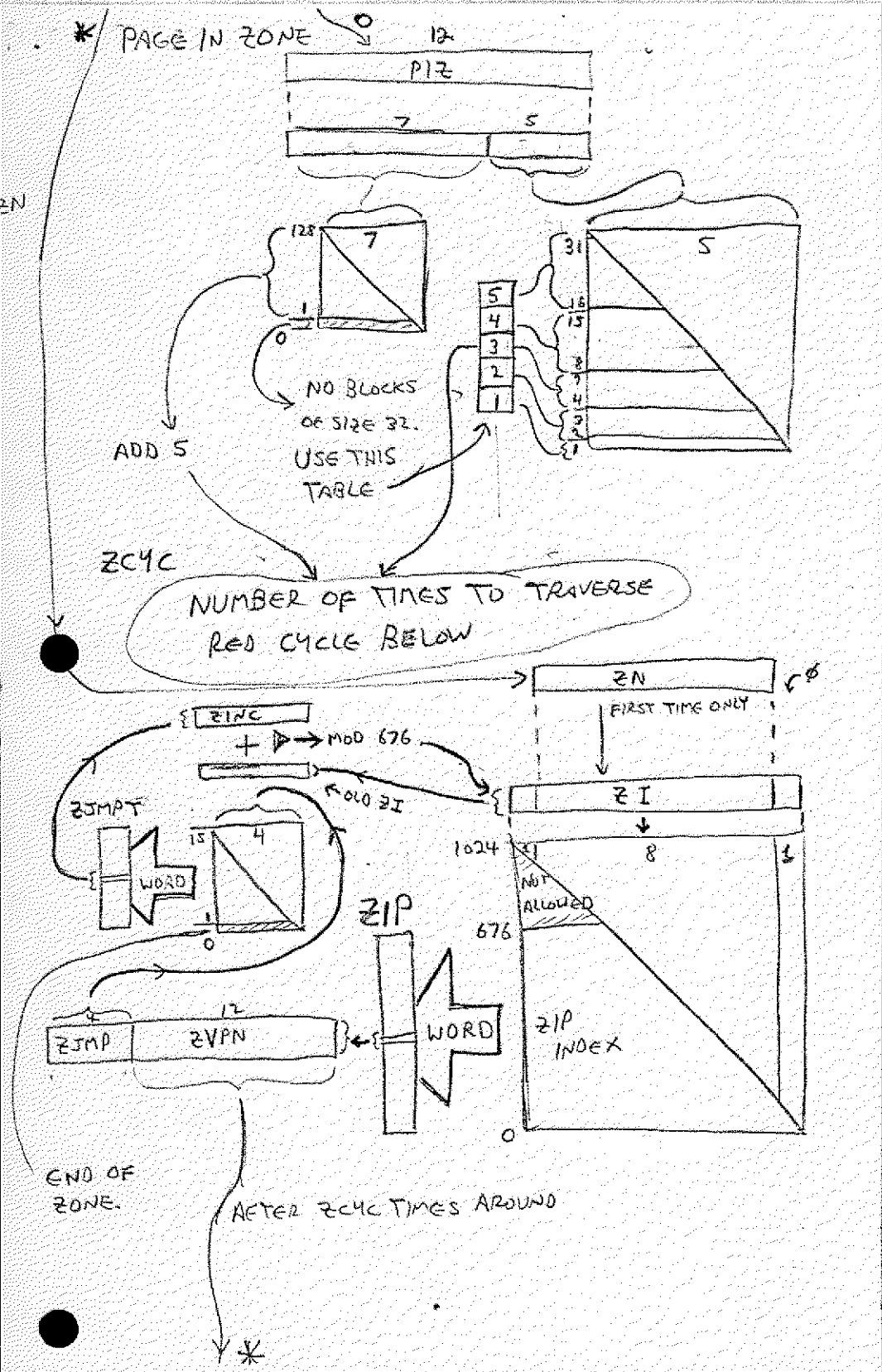
P1Z
PAGE IN ZONE
ADD 1

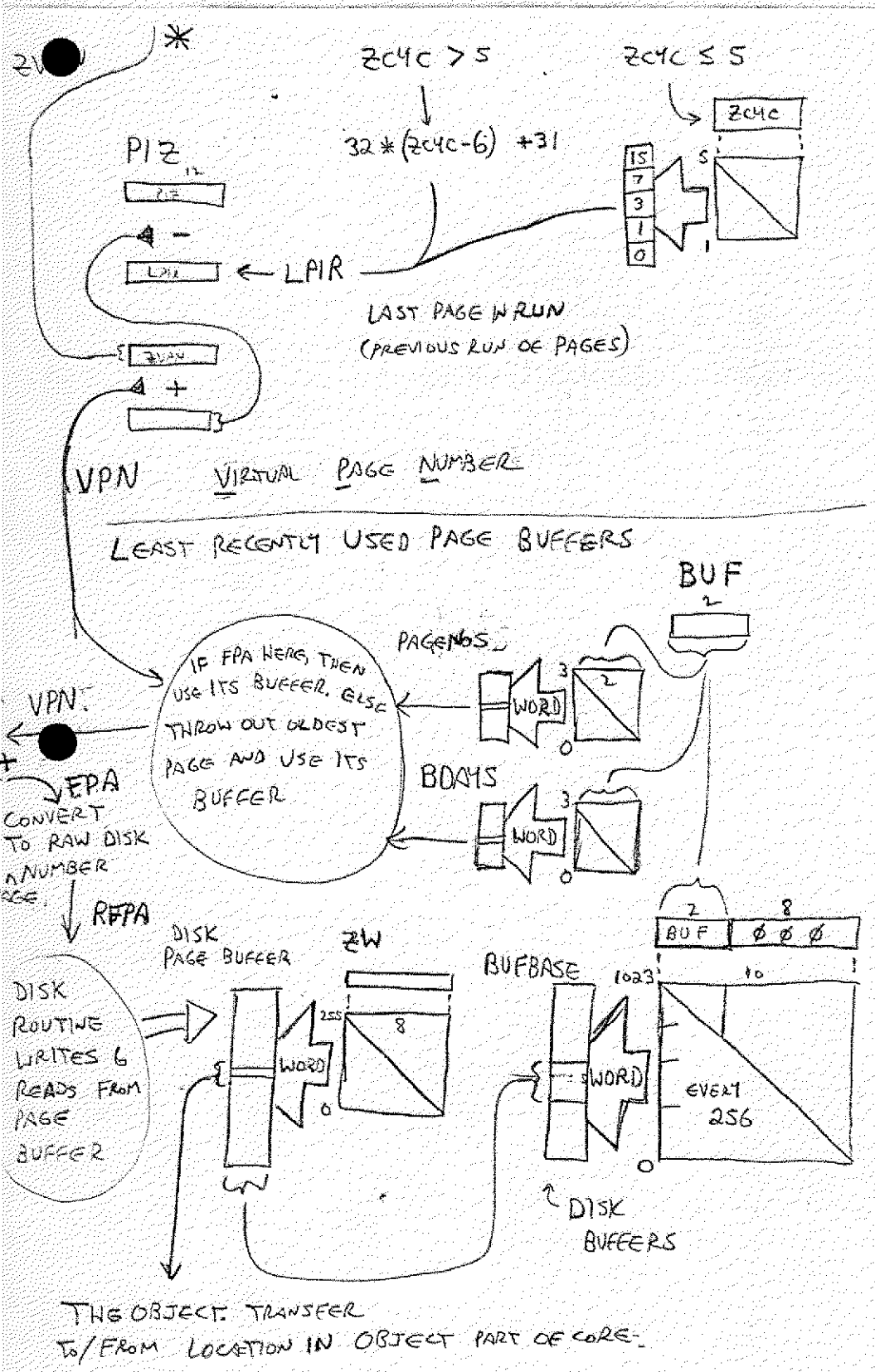
WORD OFFSET IN PAGE OF
FIRST WORD IN OBJECT

ZW

*

0





CONVERT VIRTUAL PAGE NUMBER TO
FILE PAGE ADDRESS

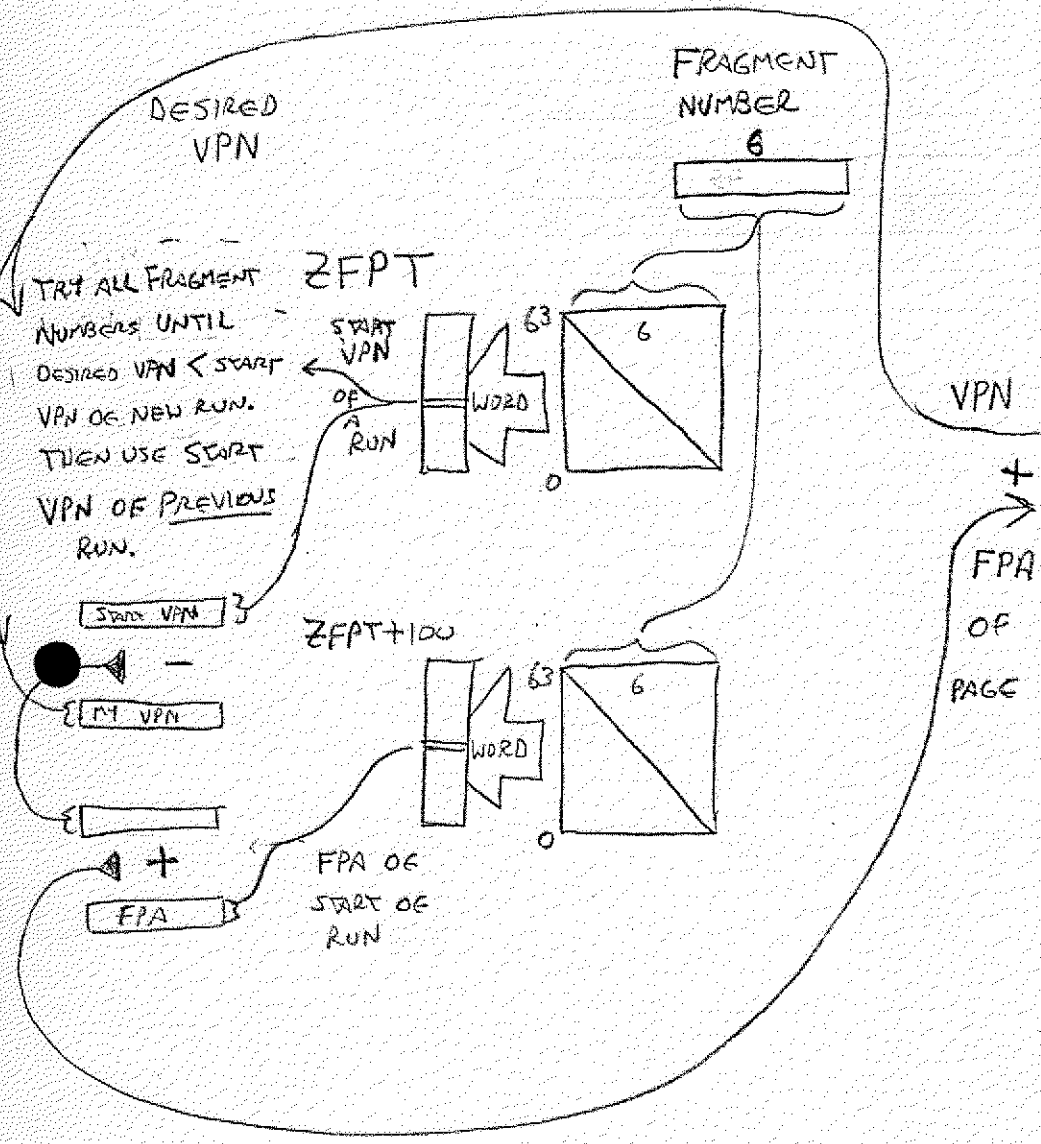
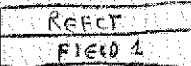
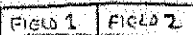
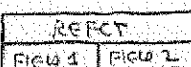
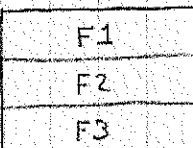
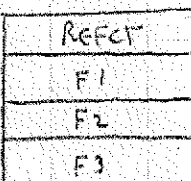
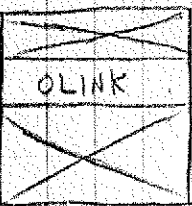
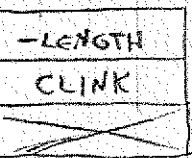
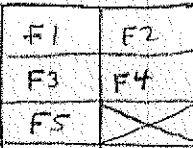
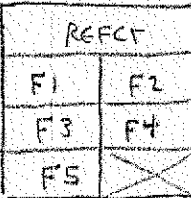
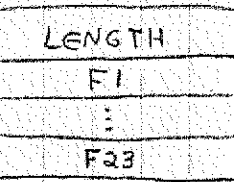
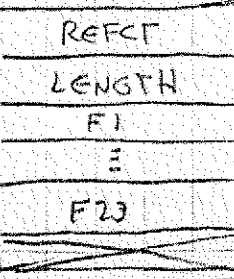
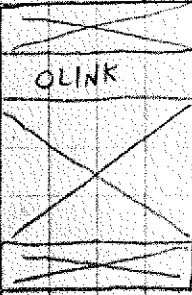
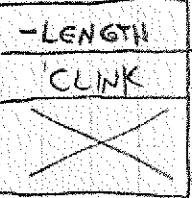
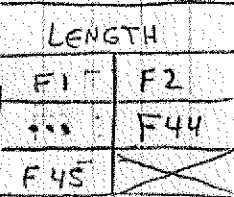
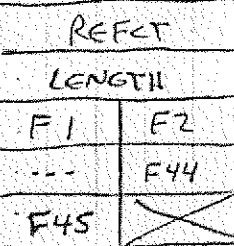
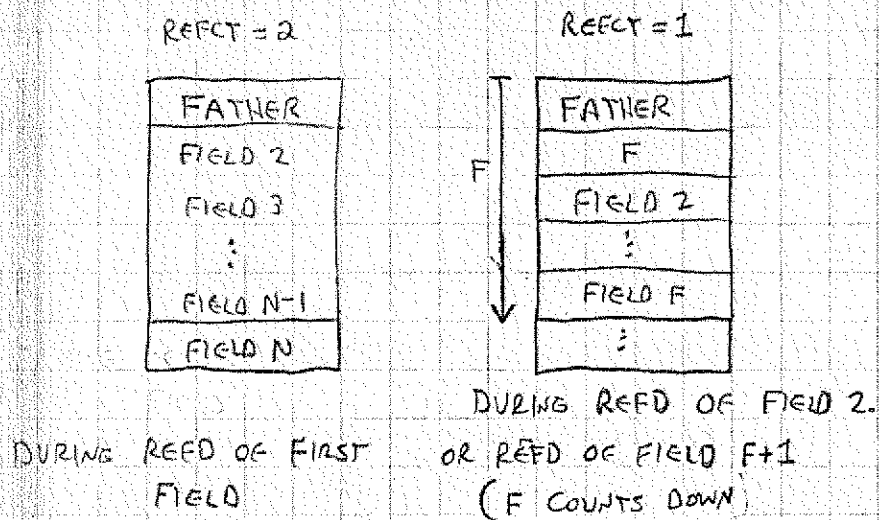


FIG 12.4c

OBJECT FORMATS		OBJECTS IN USE		OBJECTS ON FREELISTS		FREE CORE BLOCKS (CORE)
		CORE	DISK	CORE	DISK	
EXACT	IMMEDIATE DATA	WORDS ROT1: 		ROT1: 		ROT ENTRY IS FREED
		BYTES ROT1: 				
	NON IMMEDIATE DATA	WORDS → 				
		BYTES → 				
OCTAVE		WORDS → 		ROT1: 		
			①	CORE BLOCK IS FREED		
		BYTES → 			①	
			①			

① To NEXT POWER OF 9 FIELDS

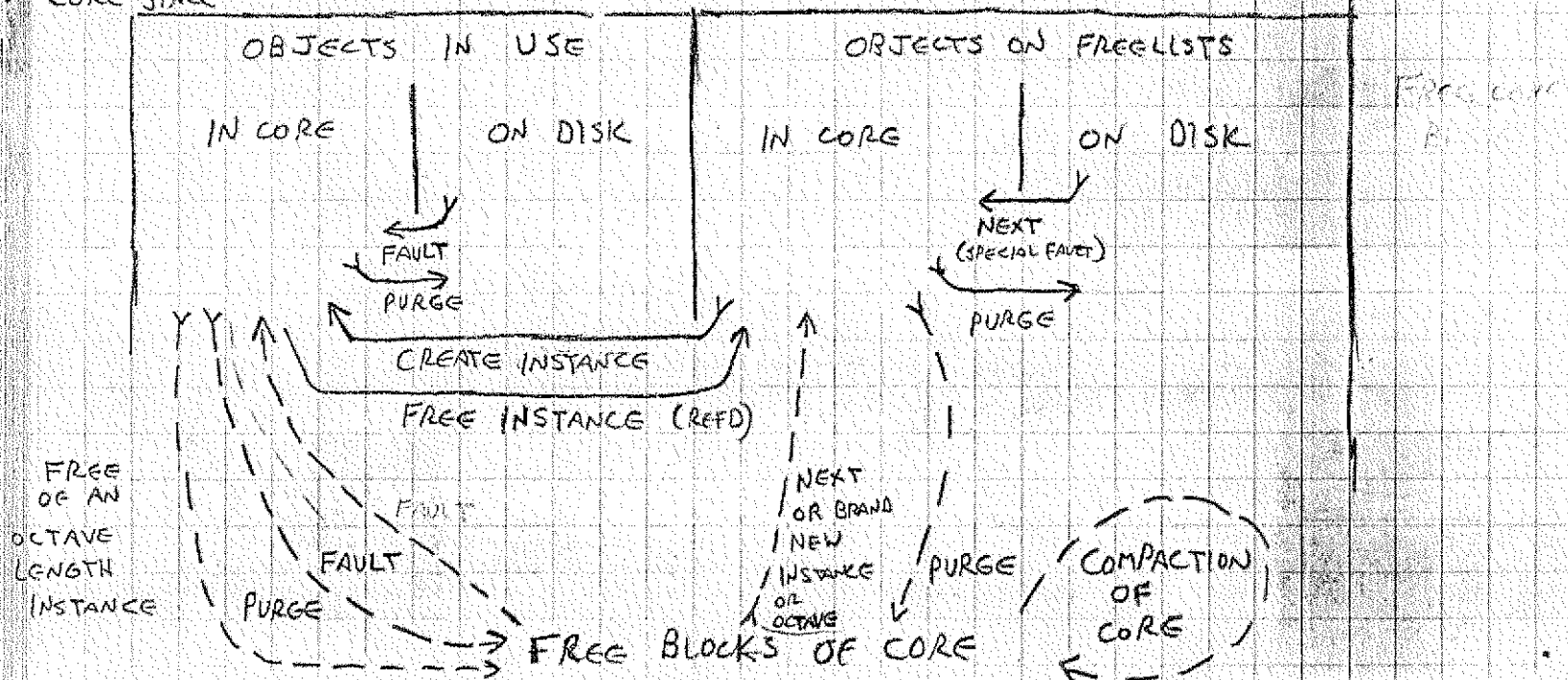
OBJECT FORMAT DURING A RECURSIVE FREE.



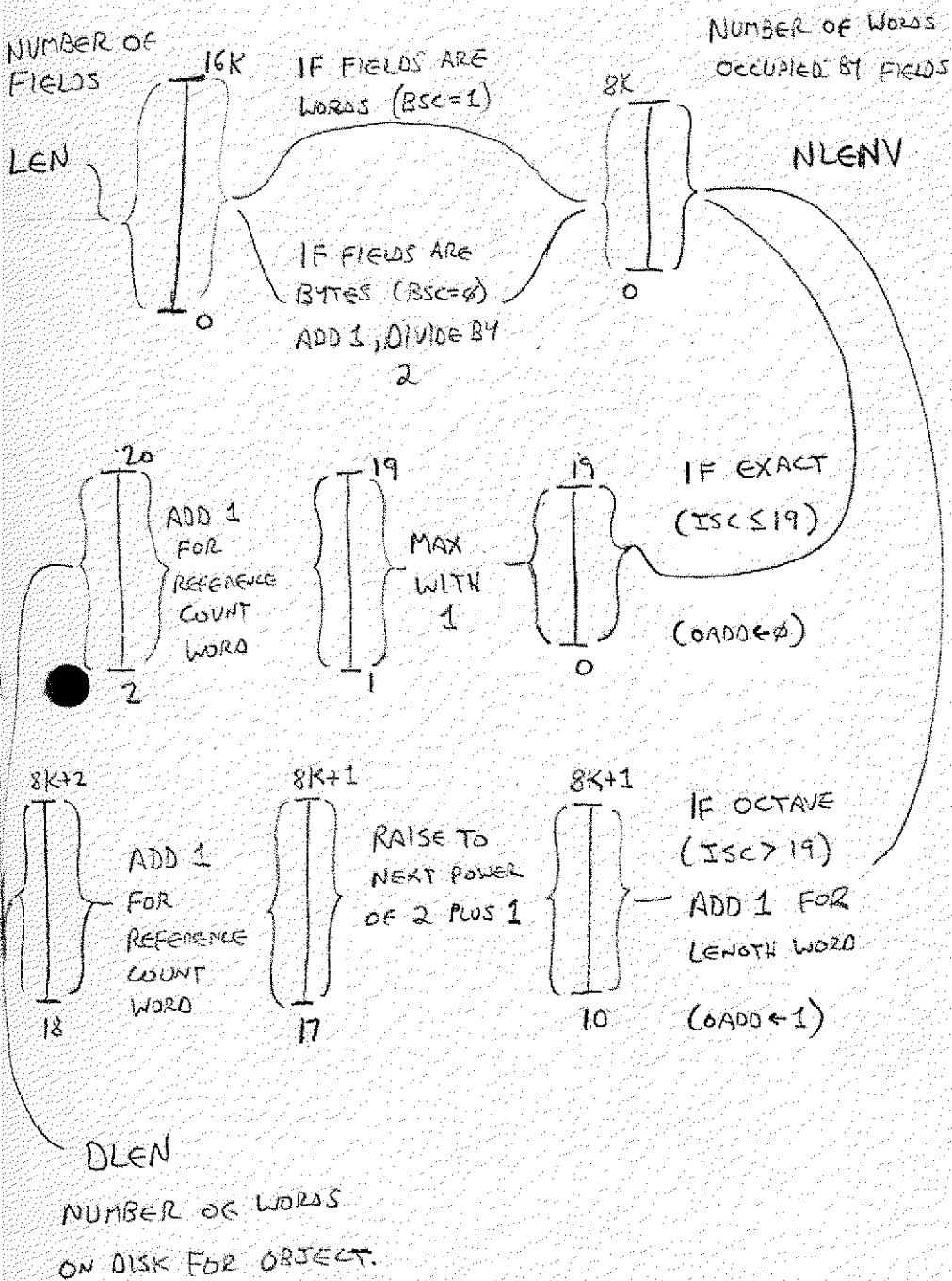
TRANSFORMATIONS OF OBJECT FORMATS

—> OBJECT ITSELF (INFORMATION)

-----> CORE SPACE



LENGTH OF AN OBJECT



REFERENCE COUNT

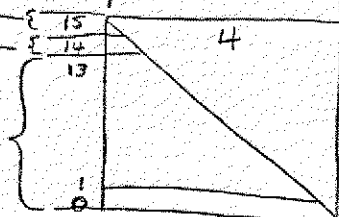
(FROM 12.4b)

THIS IS AN EMPTY ROT ENTRY

RCT

FIG. 12.4c

SMALL REF
COUNT - 1



TRUE REFERENCE

IS FOUND IN OVERFLOW
TABLE

OVERFLOW REFERENCE COUNT TABLE

AN ENTIRE WORD OF REFERENCE COUNT IS STORED
IN A HASH TABLE. A MAXIMUM OF 32 OBJECTS
WITH OVERFLOW REF COUNT ARE ALLOWED AT ONE
TIME. ADDITIONAL ONES ARE NOT PUT IN THE
TABLE, AND THEY BECOME "STUCK" AND CAN
NEVER BE FREED.

OVADR

32

REFCT

32

LINK

32

HASH

32

OOPS

COUNT

CORE
ADR.

CORE
ADR.

ALLOCATOR

NLENV
NUMBER OF WORDS OF
CORE OF INSTANCE

NLENV = 9

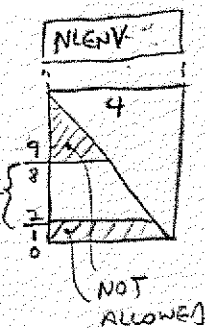
9

NLENV > 9

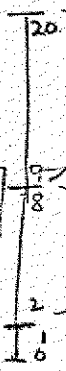
POWER OF 2
THAT
NLENV + 1 IS
LESS OR EQUAL
TO

ADD 5

NLENV ≤ 9



PTRZ



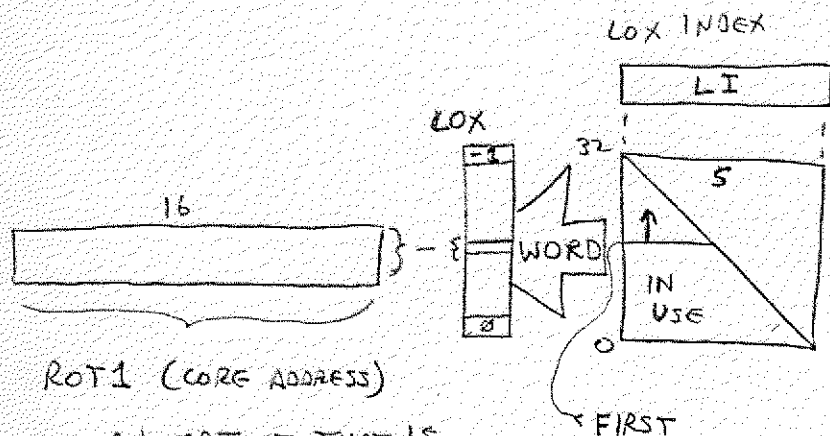
(16K+2 - 32K+1)

(9-17)

CORE ADDRESS
OF FIRST BLOCK
IN THIS SIZE
RANGE ON FREELIST
(NIL IF NONE)

LOX

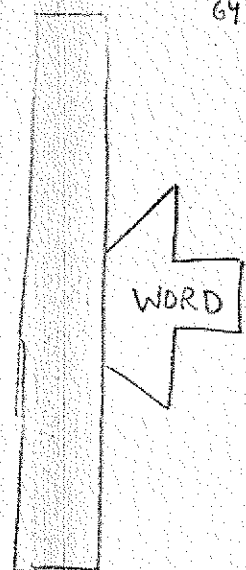
LOCKING TABLE



DURING A COMPACTION, THE LOX TABLE IS SORTED

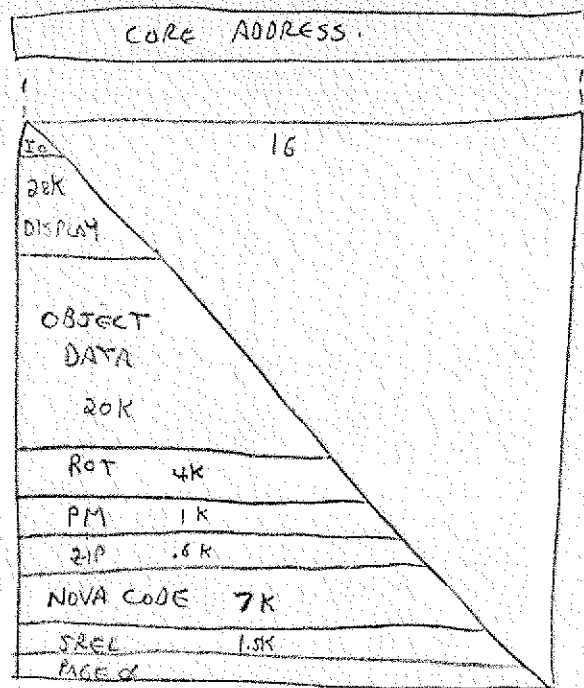
FIG 12-45

PHYSICAL MEMORY

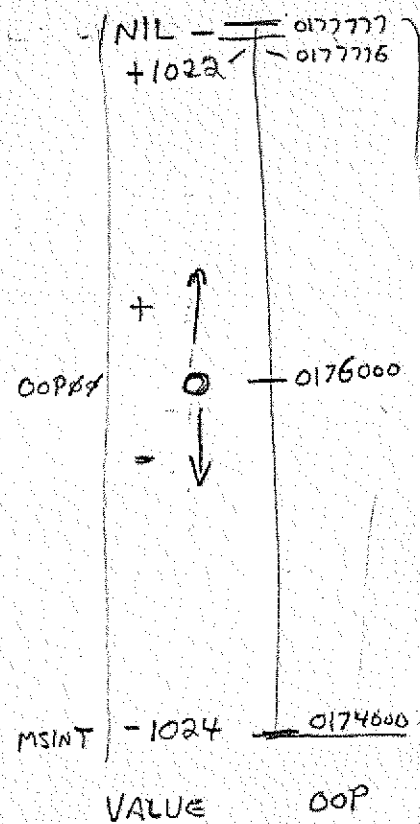


64K

0



SMALL INTEGERS.



64K

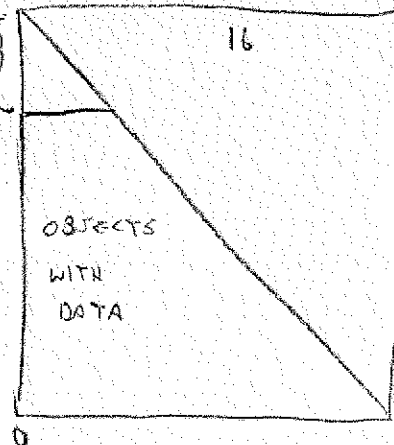
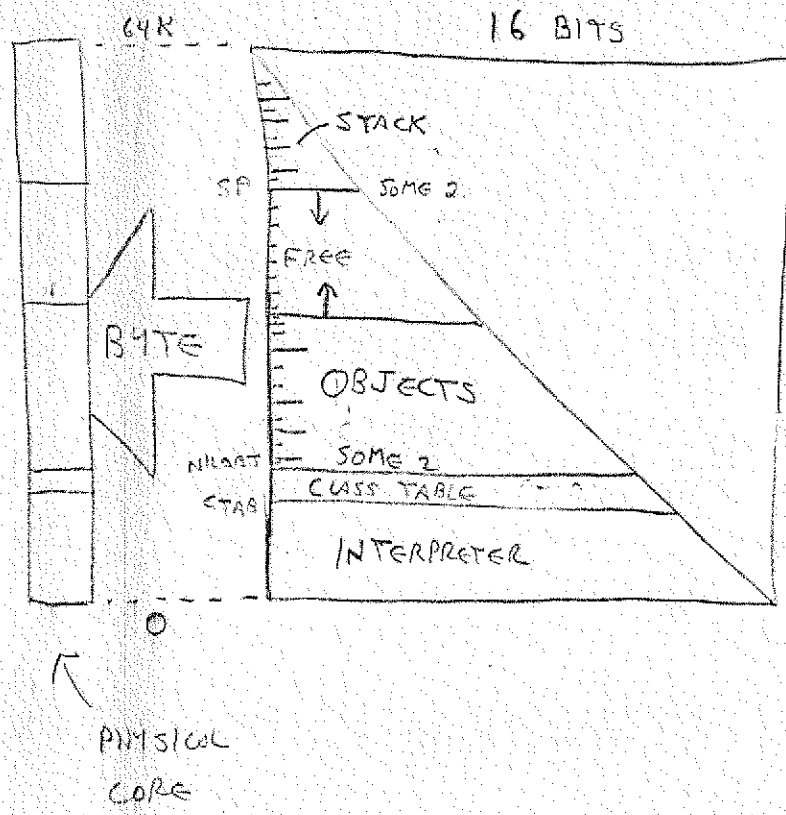


FIG 12.5a
NOTETAKER I SMALLTALK



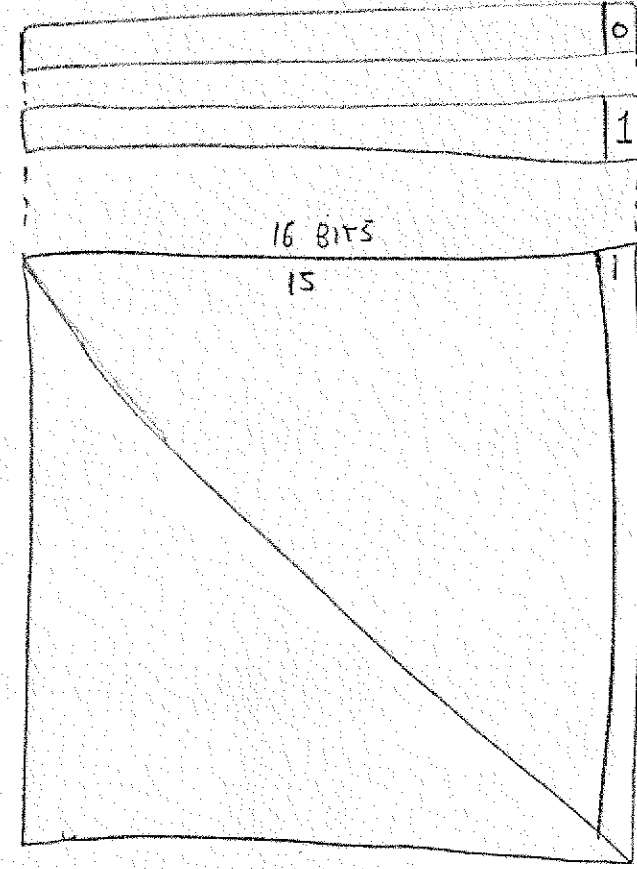
OBJECT SPACE

NORMAL
OBJECT

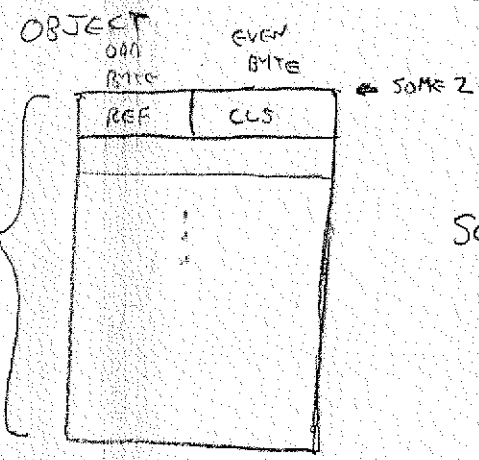
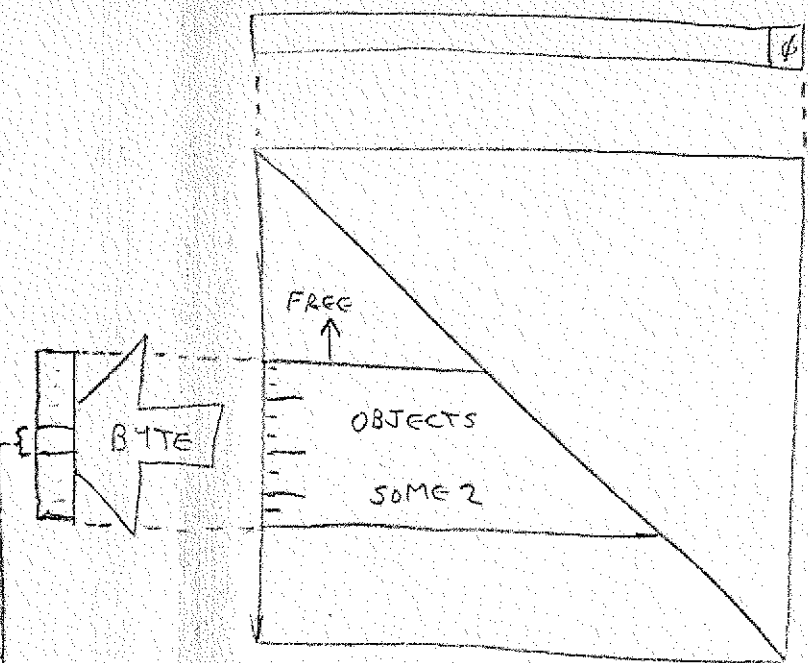
INTEGER

64K

OBJECT
IDENTIFIER



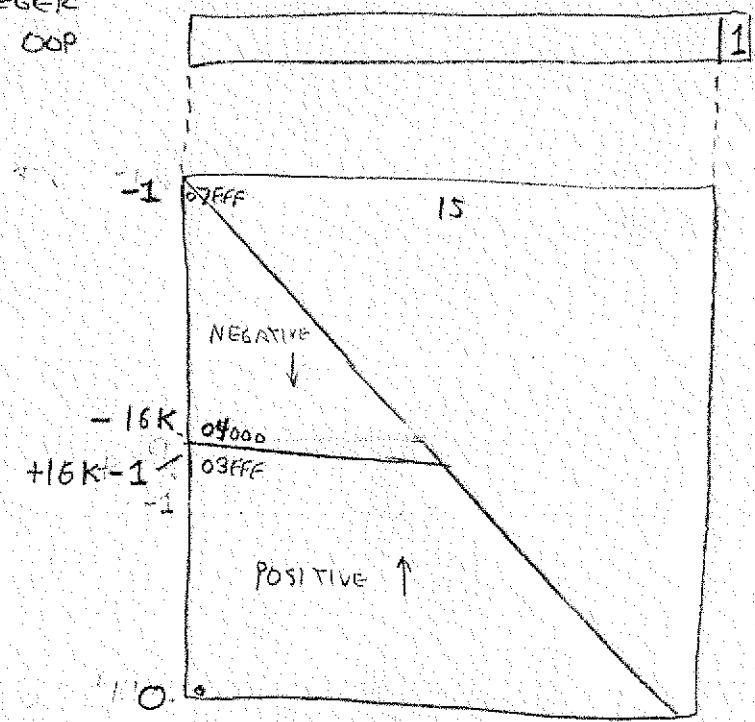
NORMAL OBJECT SPACE



SEE FORMATS OF
OBJECTS.

INTEGER SPACE

INTEGER
OOP



DEFINITION OF OBJECT FIELDS

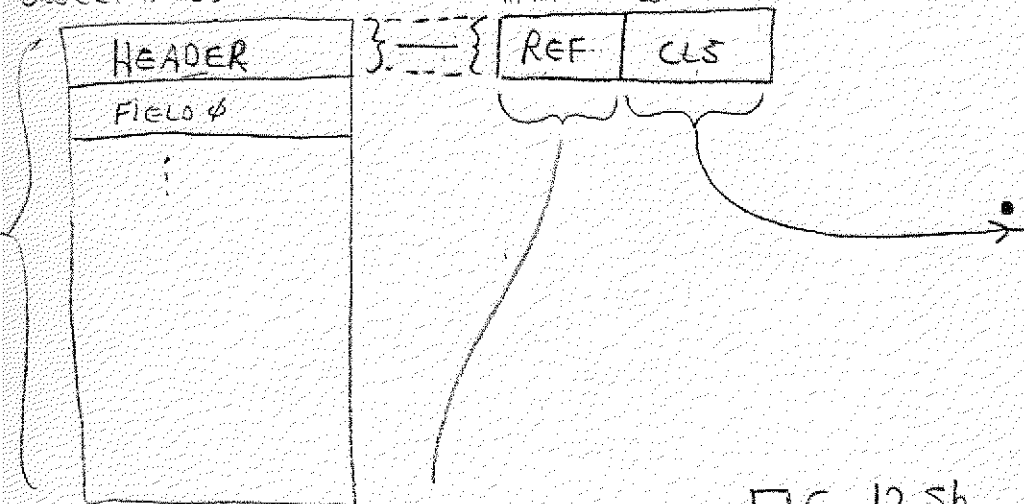
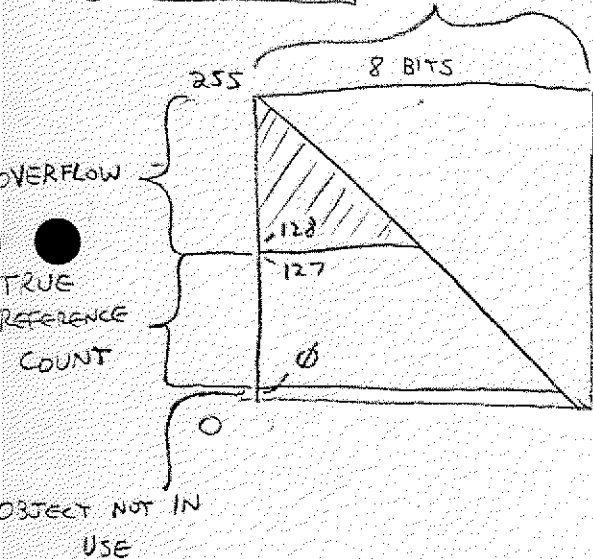
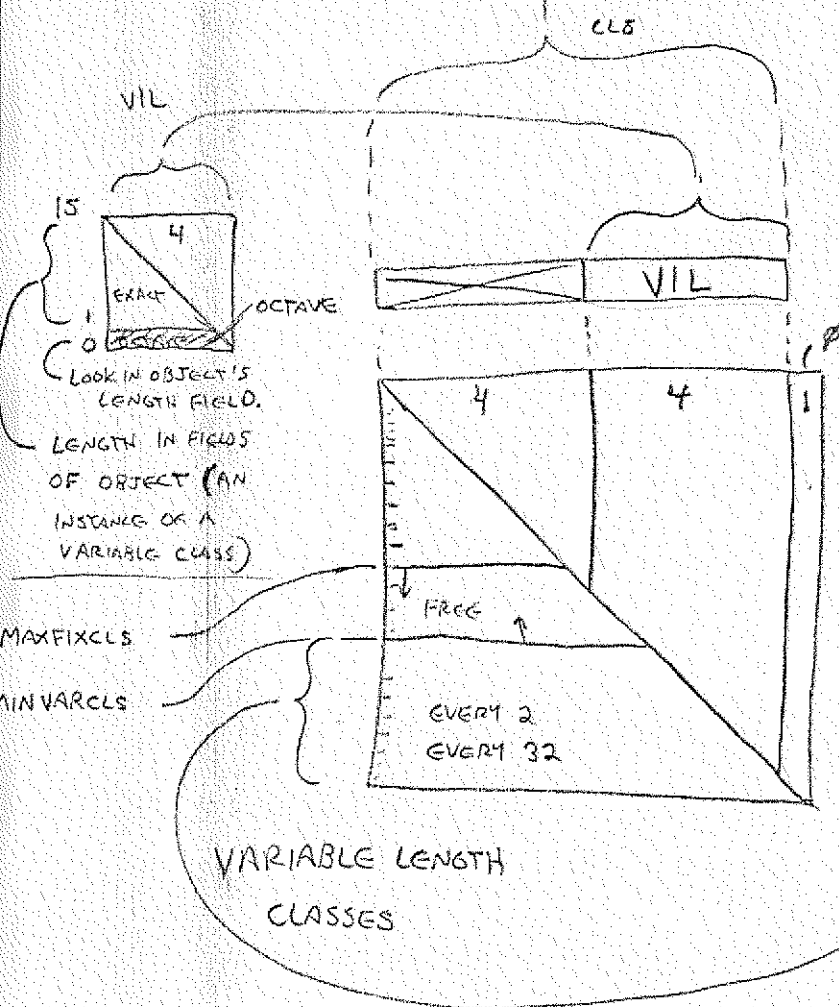


FIG. 12.5b

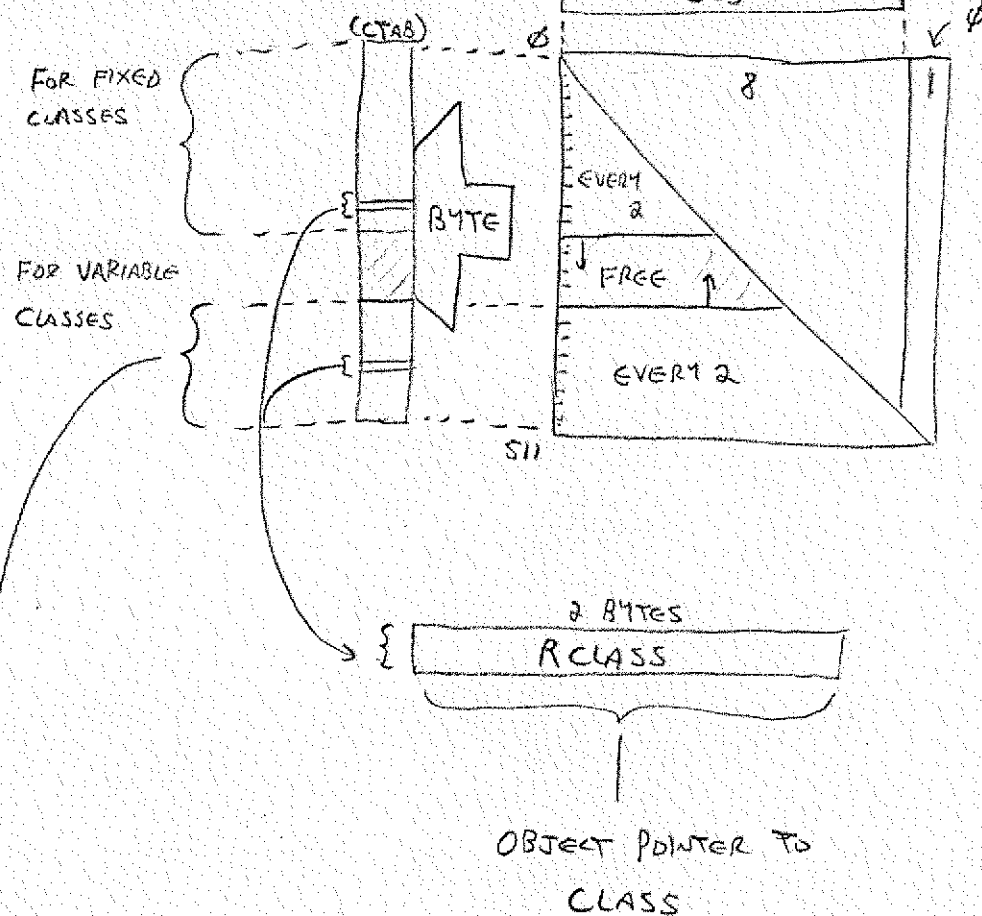


AN OBJECT WITH REFERENCE COUNT IN THE OVERFLOW RANGE IS STUCK THERE. IT WILL NEVER BE FREED. NO INFORMATION IS ENCODED IN THE CHOICE OF A PARTICULAR OVERFLOW VALUE (IT IS RANDOM).

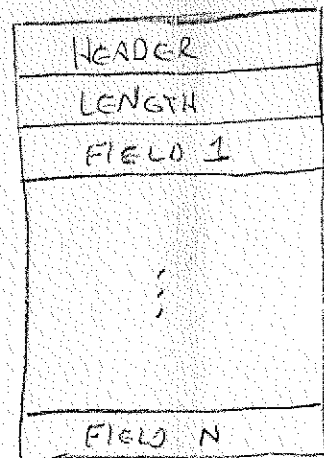
CLS



CTAB



OBJECT (INSTANCE OF A VARIABLE LENGTH CLASS)

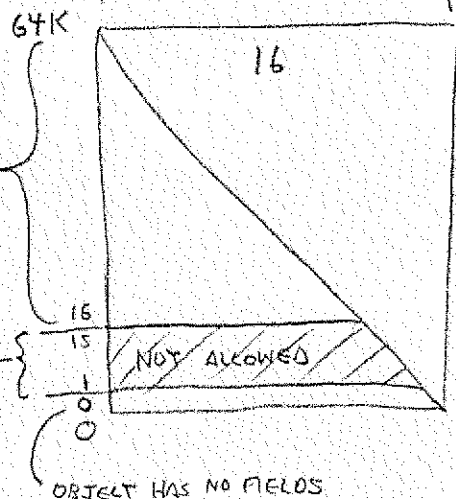


EITHER AN
OBJECT POINTER
OR
RAW BITS

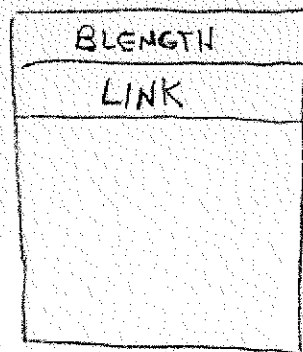
FIGURE
12.5c

NUMBER OF FIELDS
IN THE OBJECTS
(MAY BE BYTES OR WORDS)

NOT ALLOWED.
OBJECTS WITH THESE
LENGTHS HAVE DIFFERENT
FORMAT WITH NO
LENGTH WORD



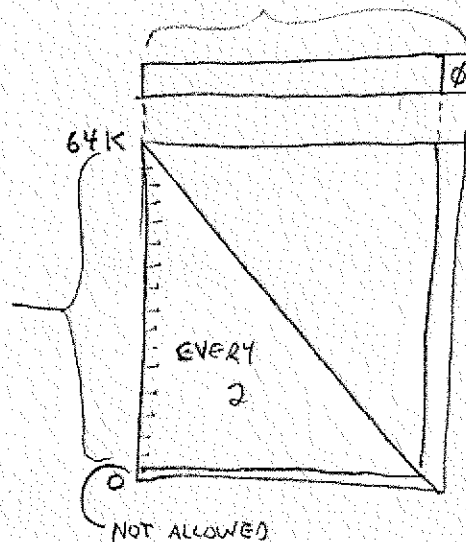
OBJECT ON THE FREELIST



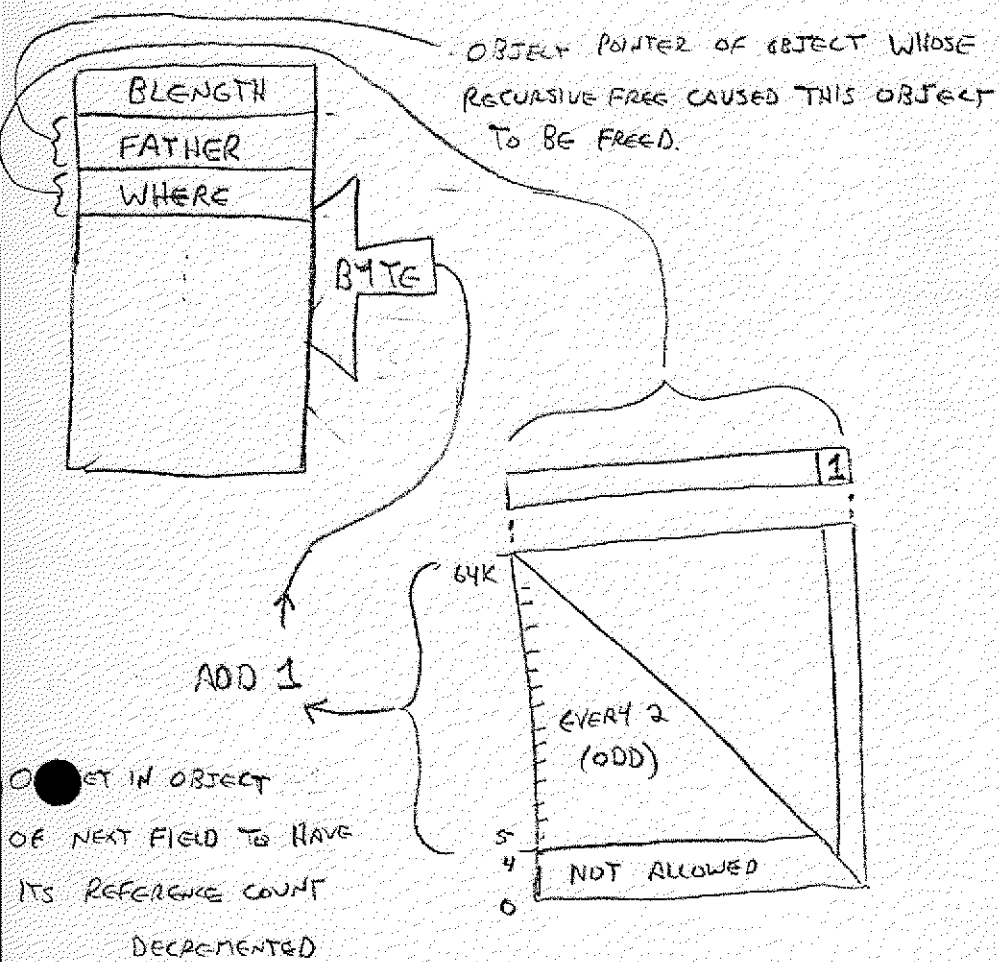
OBJECT POINTER TO NEXT
FREE OBJECT (FREE
OBJECTS
ARE SORTED BY OBJECT
POINTER)

LENGTH OF
FREE BLOCK IN
BYTES.

(ADJACENT FREE
BLOCKS ARE
MERGED AT
DEALLOCATION TIME.)



OBJECT IN THE MIDDLE OF A RECURSIVE FREE



IN USE NORMAL FIXED

REF	CLS
FIELD 1	
FIELD 2	
⋮	
FIELD n	

FREELIST

BLENGTH	(2n+2)
LINK	
⋮	

RECUE REFD FIELD 1

0	BLENGTH
2	FATHER
4	FIELD 1 IF NOT INTEGER
6	⋮
	FIELD n

1 FIELD FIXED

REF	CLS
FIELD 1	

FREELIST

BLENGTH	(4)
LINK	

RECUE REFD FIELD 2 TO n

0	BLENGTH
2	FATHER
4	WHERE : (5)
6	⋮
	FIELD n

0 FIELD FIXED

REF	CLS
NIL	

RECUE REFD FIELD 1

BLENGTH	
FATHER	

VARIABLE WORDS

REF	CLS
LENGTH	
FIELD 1	
⋮	
FIELD n	

BLENGTH	(2n+4)
LINK	
⋮	

RECUE REFD ANY

BLENGTH	
FATHER	
WHERE : (5)	
⋮	
FIELD n	

VARIABLE 0 FIELDS

REF	CLS
LENGTH	

BLENGTH	(4)
LINK	

(NO RECUE)

VARIABLE BYTES

REF	CLS
LENGTH	
F1	F2
...	...
F _n	

BLENGTH	$(\frac{n+1}{2}) \times 2 + 4$
LINK	
⋮	

(NO RECUE)

IF n GIVEN

FORMATS

HEADER	0
LENGTH	2
WORD 3	4

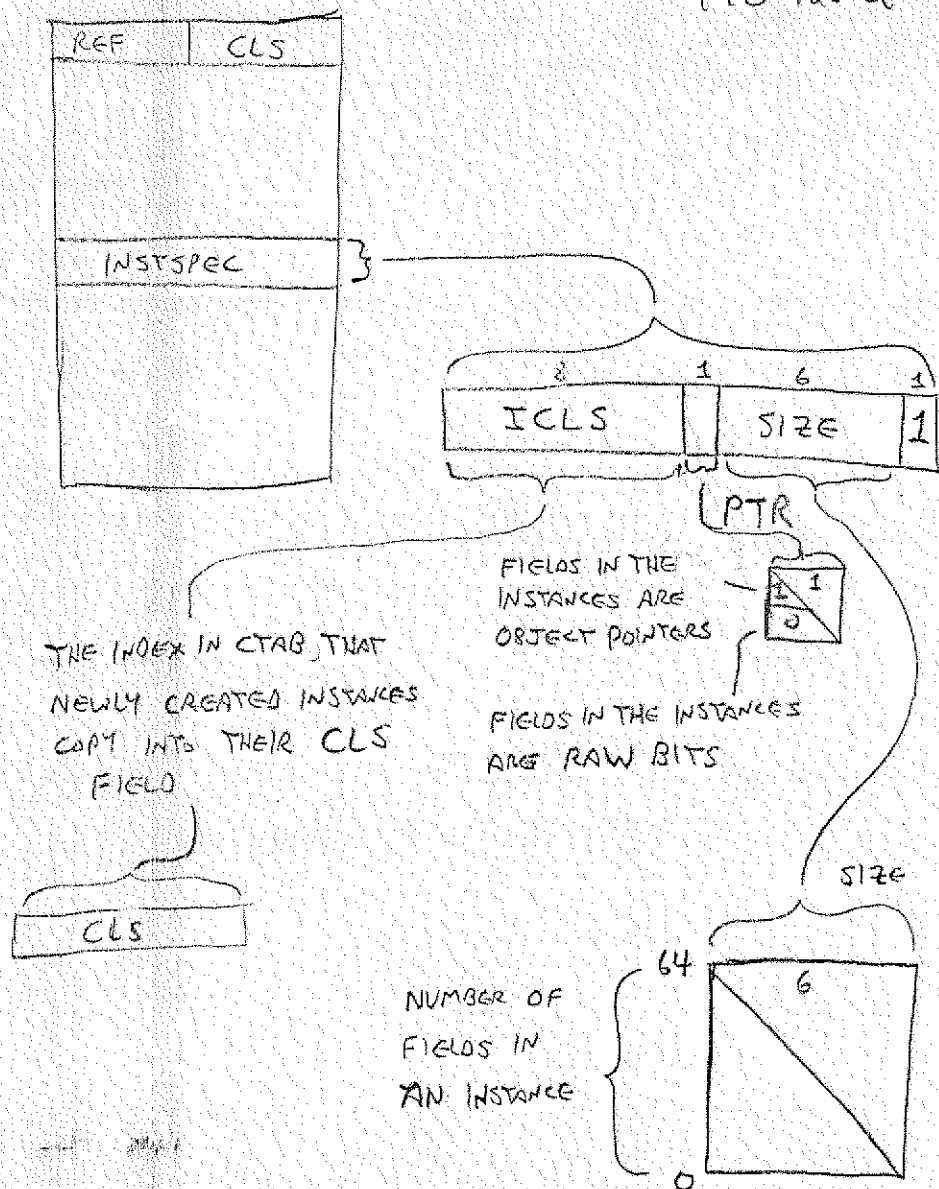
BLENGTH	0
LINK	2

BLENGTH	0
FATHER	2
WHERE	4

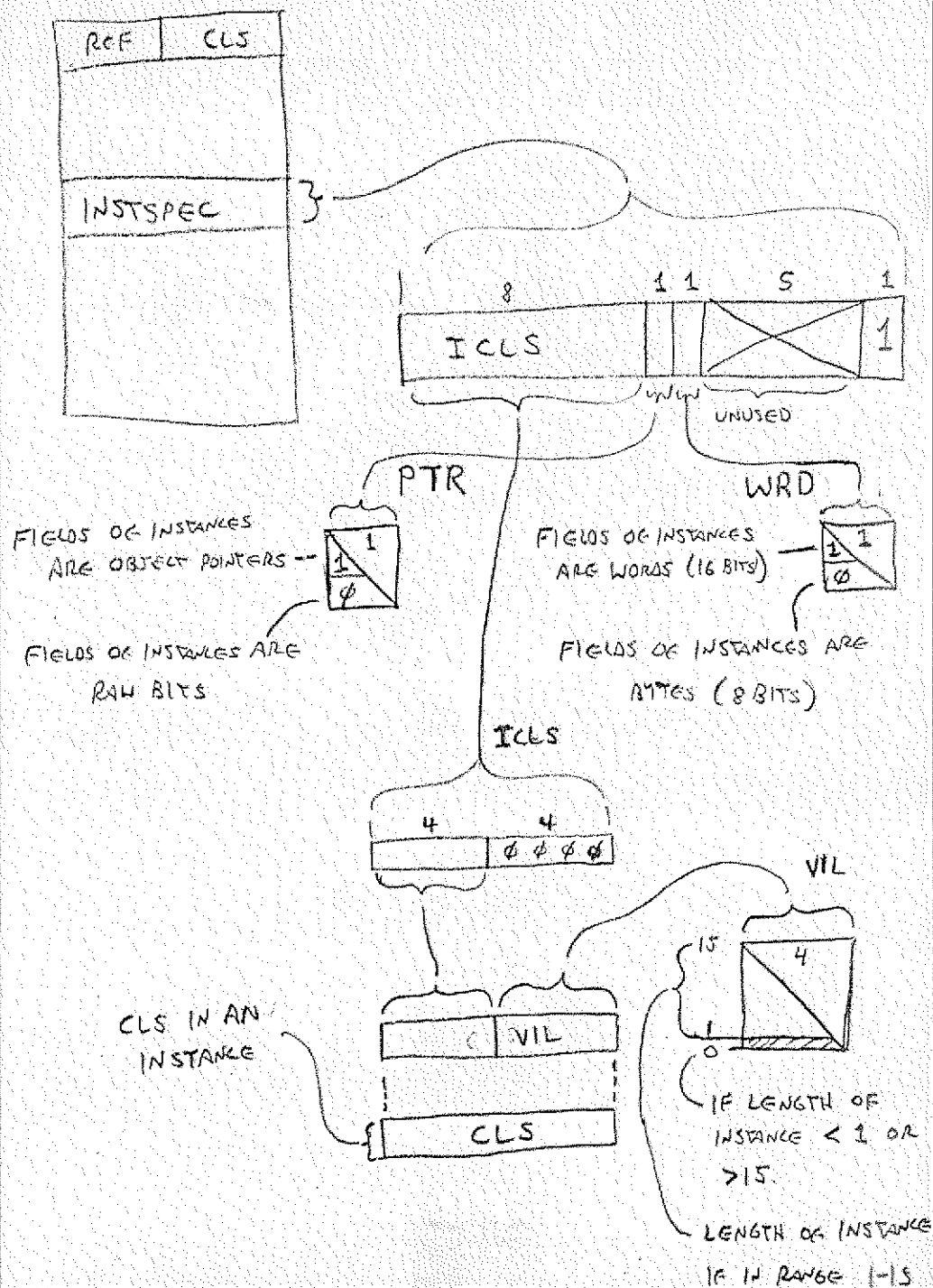
REF 0 CLS 1 = HEADER

FORMAT OF A FIXED LENGTH CLASS

FIG 12.5d



FORMAT OF A VARIABLE LENGTH CLASS



TRADE OFF ISSUES IN

VIRTUAL MEMORY DESIGN

FIG 12.6

[illegible]

COMPACT CORE

DO NOT COMPACT CORE

ELIMINATE HOLES, RADICAL CHANGE
IN MEMORY ALLOCATION.

WATCH OUT FOR FRAGMENTATION.
(NO RADICAL MEMORY USE CHANGES)

MUST RECOGNISE FREE BLOCKS.
MUST KNOW OBJECT LENGTHS.

IF HASH TABLE, CAN REVERSE IT
FOR FAST COMPACTION. MUST BE
ABLE TO UNHASH.

IF DIRECT CORE POINTERS, MUST
RENAME OBJECTS.

IF RAW CORE ADDR USED, MUST LOCK
OBJECTS

OBJECT RECOVERY

	MIXED -	PURE
PURE	* TRANSIENT OBJ NOT REECT	GARBAGE COLLECTION
REFERENCE COUNTING	* MATURE OBJ REECT	
	* CIRCULAR STRUCTURE	COLLECT EVERY K OBJECTS
TAKE $\frac{1}{2}$ COMPUTE CYCLES FOR TRANSIENT OBJ	GARBAGE COLLECT.	CREATED. MUST HAVE EXTRA K * LEN WORDS STORAGE.
NOT FIND CIRC. STRUCTURE.	IF. FAST SWEEP OF TRANSIENT, MUST RENAME OBJ AS IT MATURES.	TIME $\propto$ SIZE SYSTEM. COLLECTS CIRC STRUCTURES.
IF SMALL REECT, MUST HAVE OVERFLOW TABLE.	IF STACK NOT REECT, MUST LOG REECT $\rightarrow$ ϕ BEFORE RELEASE.	
	OVERNIGHT GARBAGE COLLECTION.	

DON'T REECT CERTAIN OBJECTS

THEY ARE PERMANENT - CAN'T DELETE

MUST CHECK QUICKLY - BE IN A LUMP AT TOP OR BOTTOM ADDR SPACE.

INDIRECT POINTERS - DEALLOCATE OBJECT OR CORE??

BREAK DOWN DEALING CORE

GOODS FOR COMPACTING CORE.

ENCODE SPECIAL THINGS IN POINTER SPACE

CORE POINTERS

CIRCUMVENT HASHING, $\frac{1}{16}$ ADDR SPACE
COVERS OBJECTS IN CORE.

IF COMPACT, MUST FIX UP FIELDS
IN CORE.

PC CLASSES

HOW ENCODE CLASS OF OBJECT?

AS A FIELD

1 FIELD / OBJ WASTED

AS PART OF POINTER

REQUIRES INDIRECT POINTERS
WORKS WELL WITH LENGTH & DISK
ADDRESS.

IF USE FIXED SIZE CHUNKS, PENALIZES
CLASSES WITH SMALL # INSTANCES.
RESTRICTS OTHER USES OF POINTER

HOW ENCODE LENGTH OF AN OBJECT?

AS A FIELD

LOOK IN THE CLASS.

WASTED FIELD / OBJ

SLOW

IF VARIABLE LENGTH CLASS USE A
FIELD OR USE PCLOSSES.

DISK VIRTUAL MEMORY		
PAGING	OBJECTS = GRAIN	OVERLAYS OR GROUP FAULTING
CORE SPARSE SOFTWARE PAGE MAP (PLUS OBJECT MAP?)	CORE PACKED WITH USEFUL OBJECTS. NO LOCALITY USED. ~ 1 SEEK/OBJ.	
	IF USE PAGE BUFFERS, GET MULTIPLE HITS (GOOD)	

[illegible]

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. On the left side, there is a large, solid black circle representing a hole punch. The paper appears slightly aged or off-white. There is no handwriting or other markings on the page.

Inter-Office Memorandum

To

Ooze Interest

Date

April 29, 1977

From

Ted Kaehler

Location

Palo Alto

Subject

Name Composting

Organization

LRG

A blend of reference counting and garbage collection

XEROX

Filed on: <Kaehler>Compost.memo

Of space we have a fixed amount,
To get it back, we reference count,
On every store we do it twice,
Which spreads the work all flat and nice,
It uses all the time save half,
So mention speed and we'll all laugh.

The other scheme speeds storing up,
But gobbles core down by the cup,
Everything gets space anew,
If only memory grew and grew,
But even that would overflow,
And collecting garbage goes soo sloww!

Cleaning core, I can't complain,
But fill the disk, and what a pain,
Between the pointing and the paste,
A half an hour could go to waste,
I'm trying to take the best of each,
Perhaps there's speed within our reach ...

In a virtual memory such as ooze, there are two kinds of objects, shortlived objects and longlived objects. Shortlived ones are created and destroyed at a tremendous rate. Each one lives but a short time. At any one time shortlived objects make up only a small fraction of the entire virtual memory. Longlived objects comprise the bulk of virtual memory. They are created and destroyed at a low rate.

The ideal garbage collector would know instinctively which objects were shortlived and which were longlived. Garbage collection of only shortlived objects would be very profitable, since they represent most of the reclaimable space (if they are even slightly old, then they are probably dead and may be recycled). If the garbage collector searched only the shortlived objects and any objects that point to them, it would return a large amount of space in a short time.

Consider a group of objects that have just passed out of youth. The majority of them are dead shortlived objects and a few are live young longlived objects. This group is an ideal group upon which to do a quick garbage collect. All references to this group of suspected dead objects comes from objects that were touched in the recent past (they were touched when the reference was stored into them). Thus a garbage collection of the young group, need only search the recently touched objects for references to members of the group.

A normal garbage collection recursively walks the entire tree of objects, marking them 'in use' as it goes. Since most of the entire tree of objects lies painfully inaccessible on the disk, one suddenly wishes one had reference counts for the group of young objects. But now is an excellent time to create reference counts for them. All references are guaranteed to lie among the recently touched objects (in core). There will be no costly traffic in reference counts since we will recursively free all dead objects and then forget completely about reference counting.

Here is the formula for freeing the dead objects among a group of suspected dead objects. Sweep over all of core constructing reference counts for all objects in the suspect group. (For each pointer to an object, check if it is within the group of suspected objects. If it is, bump that object's reference count). Then pass over all suspect objects looking for reference counts that are still zero. Start a recursive freeing with each object whose count is zero. The young objects that survive are still in use, and are by definition, long lived! (As in the current Ooze, freed objects keep their name and core in anticipation of being used again. The purging routine kills them when they get old. The long lived objects migrate to the disk and are recovered by an overnight garbage collector.)

The time consuming part of this garbage collection is determining if a pointer found in the sweep is a member of the suspected group. At worst every pointer in core conducts a binary search of the suspected objects to see if it is among them. This is much too slow. Suppose that each new object created received a pointer that was time stamped in its high order bits. All objects created in a certain time interval have pointers within a specific range. A fast piece of microcode can search all fields in core for pointers within the range. When the sweep of core finds a pointer within the range, it hashes to get the object and bumps the reference count.

A pointer now has this form: <Time created>,<other info>. Since most new objects are shortlived, the time part of the pointer must run circularly and be used over and over. But what about the few new objects that survive to become longlived? How do we know not to use their pointers again when time has gone a full circle? Keeping and consulting a list of all longlived objects is too costly.

If new pointers are created in a portion of the address space reserved entirely for new objects, then the survivors that become longlived objects must be renamed. To rename an object, we must find every reference to the object and change it. Fortunately, the sweep of core to create reference counts finds all references to new objects. The sweep can replace every temporary new pointer with its permanent name.

A pointer looks like this:

| VVVV | PP | IIIIII | SSS | = 16 bits

VVVV = Volume number of permanent pointers. New pointers are 017. Thus one sixteenth of the address space is reserved for shortlived objects.

PP = Phase of time. Four phases allow creation in the first phase, aging for two and recovery during the fourth. (Each phase starts its group of names through this process).

SSS = Size info (a part of the pseudo class number). Eight gradations of size provide some information for finding an object on the disk.

IIIIII = Instance number. Only 128 objects of a given size may be created during one phase.

How do we assign permanent pointers to those new objects which survive after the fourth phase? We have complete freedom in assigning an pointer except that the size field must remain the same. Each of the eight sizes simply uses up names in order and keeps track of the latest pointer used.

Will name composting pay for itself?? Here are some numbers I would like to know:

1. Suppose the cost of reference counting in the current Ooze were magically reduced to zero. How much working core space could Ooze lose and still preform better than now. The space would go for objects waiting to be recovered.
2. The cost of recovering a dead new object can be divided into two parts. One cost is associated with the creating and freeing of the object. Another is the overhead cost of

sweeping core. The relative cost of the sweep overhead determines how often core may be swept and how few objects wait for recovery.

3. The number of reference counts is proportional to the number of stores in pointer fields. Thus we would like to know the ratio of pointer stores to object creations.

Inter-Office Memorandum

To SMALLTALKERS

Date May 13, 1975

From Dan Ingalls

Location Palo Alto

Subject OOZE

Organization LRG

XEROX

*** DRAFT ***

This is an incomplete document, but here it is, in case you have suggestions for how it should be completed

CONCEPT

OOZE is an object oriented virtual environment. This means that a pointer in OOZE designates an object, not an address. A specific implementation with 16-bit pointers can therefore handle 65536 objects. If the mean size of an object is 20 words, this amounts to addressing an entire model 31 disk.

Figure 1 illustrates the organization of OOZE. References to data are made indirectly through the master object table (MOT). This achieves expansion of the address space and flexibility of data management at the expense of an extra level of indirection on all references.

Entries in the MOT must at least locate the data (disk page, word offset), but may include other more descriptive elements such as size and reference count.

The chief design issues involved in an implementation are:

- Choice of descriptor (MOT entry) format
- Management of MOT and DATA
- Resident form of MOT and DATA
- Management of resident objects
- Swapping structures and methods

FIRST IMPLEMENTATION

The first implementation of OOZE will be as the foundations of the new SMALLTALK system. This slightly complicates the final debugging environment, but provides the most interesting test conditions I can imagine.

USER INTERFACE

OBJ ← CREATE(LENGTH) will allocate an object with room for LENGTH words of data.

REFI(OBJ) will increment the reference count of OBJ.

REFD(OBJ) will decrement the reference count of OBJ. If the resultant count is zero, the associated storage will be freed.

VAL ← LOAD(OBJ,WORD) returns the WORD-th word of OBJ
STORE(OBJ,WORD,VAL) stores VAL into the WORD-th word of OBJ.

PTR ← LOCK(OBJ) returns the core address where OBJ is stored, and prevents OBJ from being moved thereafter.

UNLOCK(OBJ) undoes the restriction imposed by LOCK, and invalidates PTR.

The first five of these are needed to support the SMALLTALK interpreter, and the last two will be necessary for certain structures such as fonts, timbres, ether buffers.

FORMAT OF THE MOT ENTRIES

Disk address: 14 bits are sufficient to address all of a Model 44 disk.
Word offset on page: 6 bits are sufficient here, as allocation will be in chunks of four words.

The MOT entries will be 1.5 words each, with the extra bits remaining available for such flags as may prove useful in the application. Favorite candidates so far are: monitor, lock, and pointer(meaning this object is all pointers).

FORMAT OF DATA

Data blocks are allocated in units of four words on the disk. The first word of every block contains an 8-bit reference count and an 8-bit length. If ever the refct or length overflows, then their full value is found by reference to an escape table. A reference count of zero denotes a free block, and such a block will carry forward and backward links for allocation (and possibly a size extension).

In favor of locating length and refct with data is the in-core situation where the disk pointers will be thrown away, but the length and refct are still needed with the data. Also, the attractive boundary-tag allocation scheme wants a length field and free tag in every reserved block.

MAINTENANCE OF DATA

The reservation and liberation of data blocks will follow Knuth vol 1 ex 2.5.19. However, a workingset effect might be encouraged by trying to pick up the free list near a recently referenced or allocated object. Scanning to find the freelist and scanning along the freelist should be possible in the shadow of chained disk reading.

ATOMS AND SMALL INTEGERS

Atoms and small integers will be recognized as having object numbers within a certain range, say atoms 0-3777, small integers 4000-4777. Atoms being thus recognized, may simply point to their pname strings in the data file. Small integers betray value as well as class by their object number, and hence need no entry in the MOT or DATA files.

Atoms do not carry reference counts - garbage collection of the atom table is a slow and painful process. - Do it overnight along with detection of circular structures.

RESIDENT OBJECT TABLE (ROT)

The ROT will be a hash table of about 1000-2000 entries. It must store the core address of each resident object, and also the object number key residue, a lock bit for objects which must not move, a dirty bit, a vacant bit and a reallocate bit (see below), and some aging information. It looks like a two word entry.

One might be inclined to save the disk pointer in the ROT, but this is a high space penalty. Moreover, it will only be of use for data which becomes dirty and has to be written out, probably only 10 percent or less of the transactions. This figure may seem a low estimate, but remember that we only need to worry about those objects which really become dirty, not all those which happen to be on the same page as another which happens to get dirty.

MAINTENANCE OF RESIDENT OBJECTS

Resident objects find themselves in just the right format for an in-core version of the same allocation scheme as is used on the disk, except that the free chain links will be core pointers instead of disk pointers!

There should be a mechanism for marking a resident object "vacant" if it has been freed, but has not yet been written out onto the disk as free. This would allow both its object number and resident storage to be reused without core shuffling or disk allocation. To this end, the ROT entries must include a "vacant" bit and a "reallocate" bit. The latter means that, since this object number has switched to designate a block of a different size, the previously designated block of the DATA file must be freed, and one of the new size located and installed in the MOT.

The reallocate bit should allow for trivial data file compaction: one simply reads in an object and turns on its reallocate bit. It will then have its old self freed, a new copy allocated (presumably in a more appropriate place), and the MOT will be duly updated.

The recursive reference counting which assures freeing of substructures all takes place in the resident space. Freeing of objects on the disk will be simple freeing.

DECISIONS SO FAR

Implementation plan above
User Interface as above
Data will be allocated in 4-word chunks.
Length and reference count reside in the data blocks
Master object table entries are 1.5 words = <disk addr, word offset, flags>

UNSPECIFIED AS YET

Swapping algorithms - a juggling act of reference counting, maintaining dirty bits, and incorporating the 'vacant' object concept.
Swapping strategies to prevent many seeks to same page when dealing with grouped data.

REAL QUANDRIES

Just how much should OOZE know about the data it stores? It seems that it will have to know whether an object is composed of pointers in order to maintain reference counts. Should it also try to figure out size information from the class pointer?

And how much should the interpreter know about OOZE? Atoms and small integers are two examples where separation breaks down.

(In order that SMALLTALK strings and vectors (etc) be stored with reasonable efficiency, it would be nice if, for instance, the additional 3 bits which specify byte length (finer than block length) could be encoded into the class pointer, say by having 8 flavors of class string. This is not necessary at the outset, but if not done, two bytes of string is all that will fit in a four word block of disk space (not counting MOT entry)! It would be possible to convert to the normal form (length word follows class pointer) at the time a string is brought into core, and vice-versa, but it's another case of the bottom level routines needing higher-level smarts.)

DO

NOT

TOUCH

pen

Many (blank ^{from} 10 to 127 what how to foo

penup
pendu

CARD QUEST

RACB.OTTY

CARDQUEST.OTTY