1993 International Workshop on Object-Orientation in Operating Systems
(I-WOOOS'93), Asheville, North Carolina, Dec 9-10th, 1993

A Persistent System in Real Use
— Experiences of the First 13 Years —

Jochen Liedtke

German National Research Center for Computer Science
GMD SET
53757 Sankt Augustin
Germany
jochen.liedtke@gmd.de

Abstract

Eumel and its advanced successor L3 are operating systems built by GMD which have been used, for 13 years and 4 years respectively, as production systems in business and education. More than 2000 Eumel systems and 500 L3 systems have been shipped since 1979 and 1988. Both systems rely heavily on the paradigm of persistence (including fault-surviving persistence). Both data and processes, in principle all objects are persistent, files are implemented by means of persistent objects (not vice versa) etc.

In addition to the principles and mechanisms of Eumel/L3, general and specific experiences are described: these relate to the design, implementation and maintenance of the systems over the last 13 years. For general purpose timesharing systems the idea is powerful and elegant, it can be efficiently implemented, but making a system really usable is hard work.

1 Historical background: Eumel and L3

When the first 8-bit microcomputers became available in Germany, GMD started an effort to introduce them, as small workstations, into schools and universities. Since there was no really usable operating system for these machines (only CP/M and similar control programs), GMD and the University of Bielefeld decided to develop a new system from scratch.

Design and implementation of the Eumel (pronounced 'oimel') operating system started in 1979. The initial hardware base was a computer with a Zilog Z80 processor, 64 KB of main memory and one or more 8″ floppy disk drives storing 300 Kbytes each. Later, more memory and a hard disk were added. In the following years the system was ported to many different machines based on Zilog Z80 and Z8000, Motorola 68000 and Intel 8086 processors.

Unfortunately 8-bit and the early 16-bit processors did not provide hardware to support a state of the art operating system, in particular they lacked a memory management unit. We, therefore, designed a virtual machine with a powerful instruction set and virtual 32 bit addresses. Although the instructions and the MMU had to be implemented in software, the system's overall efficiency was high enough that commercial sites had up to 5 terminals per system.

Due to its non-standard architecture Eumel was initially a one language system based on ELAN [Hom 79]. Later, some other compilers became available (CDL, Pascal, Basic, Dynamo) but these were not widely used.

In due course processors came up with the necessary hardware support for data security and virtual addressing; the requirements of the virtual machine could then be met directly by hardware. In 1987 we started the L3 development, principally aimed at achieving higher performance and obtaining an open system. The L3 architecture is an advancement and a generalization of the Eumel principles, but built completely from scratch. Since L3 is upwards compatible with Eumel, it inherited all of the existing tools and applictions.

Both L3 and its predecessor Eumel are pure μ-kernel based systems relying heavily on the idea of persistent processes. They are strongly influenced by Multics and have similarities to later systems, such as Accent and Mach.

The first Eumel systems were shipped to end users in 1980, and were mainly used for teaching programming and text processing. In the following years commercial systems to support lawyers, and other specialised applications for small and medium sized companies, were built on top Eumel. By the mid 80's more than 2000 systems had been installed.

Delivery of L3 began in 1989 and, to date, about 500 L3 systems have been shipped. Most of these are used for commercial applications, others as Eumel successors in schools. L3 is now in daily use in a variety of industrial and commercial contents.

2 Related work

The Eumel/L3 model of virtual memory was strongly influenced by the Multics [Ben 72] ideas. Lorie [Lor 77] introduced shadow paging for checkpointing. A slightly more general variant of this method is used in Eumel/L3 for both copying and checkpointing.

In 1979, most related work was yet to start (or not yet widely known). Accent [Ras 81] and its successor Mach [Acc 86] used copy on write techniques. These and various other systems (e.g. Amoeba [Mul 84], Chorus [Gui 82], V [Che 84] and its predecessor Toth [Che 79]) are based on the message passing paradigm, but not on the persistence paradigm.

The programming languages Elle [Alb 80] and PS-Algol [Atk 82] already handled persistent and transient data uniformly. To date, partial data persistence (without dealing with faults) is part of the the Comandos [Cah 93] project. Data and process persistence including faults is also supported in Monads [Ros 87] and by the KeyKOS [Har 85] nano-kernel, first released in 1983.

The Eumel/L3 concepts and experiences have also influenced the BirliX [Här 92] operating system design at GMD.

3 Principles

3.1 Everything is persistent

We did not find any conceptual reason for anything *not* to be persistent. Obviously all things should live as long as they are needed; and equally obviously turning off the power should neither be connected with the lifetime of a file nor with the lifetime of a local variable on a program stack. There are files that are only needed for a few seconds, and programs that run for weeks.

Some data (e.g. files) have to be persistent, so we decided that all data should be persistent. The basic mechanisms used to achieve this are *virtual address spaces* and *mapping*. Benefits of this decision are that:

- Only one general mechanism has to be designed, implemented, verified and tuned for files, programs and databases.
- New persistent data types can be built efficiently using this mechanism.
- A single, general, recovery mechanism can be constructed.
- Neither syntactical nor run time overhead is incurred when accessing persistent data.

Since data is persistent, the active instances operating on the data should be persistent too; so we decided that processes would also be persistent. In fact this only requires that stack pointers, program counters and other internal registers be regarded as data. Consequences of this decision are that:

- Program and system checkpoints are easy to implement.
- All running programs automatically become daemons.
- For more complex functions (e.g. protection and synchronisation) active algorithms can be used in place of conventional passive data structures.

Thus *tasks* — consisting of an *address space*, one or more *threads*, and a set of data objects mapped into the address space — are principally persistent.

Unfortunately not all tasks can be made persistent. When beginning the L3 design we decided that all device drivers had to be user level tasks. This idea is natural, since there is nothing exceptional about a device driver, and it proved to be flexible and powerful. But, for hardware related reasons, most device drivers must not be swapped out, or even blocked for long enough to write back their status to the persistent store. So tasks can be declared to be resident, which also means non persistent.

3.2 Processes are first class objects

Most operating systems introduce passive data as first class objects: they give unique identifiers or even path names to them, permit low level access control etc. However, when processes are also persistent, this turns out to be upside down. An active process is more general, flexible and powerful than a passive data object (although the models are Turing-equivalent), especially when implementing objects with inherent activity: e.g. traffic lights, floppy disk drives (turning off the motor if no request for 3 seconds) or a terminal handler which cleans the screen and requests new authorization (like xautoolog), when no keystroke occurs for some minutes (or the terminal camera no longer "sees" the user). A simple example for this type of object is a clock which shows the actual time on the screen and can be switched to different local times concurrently:

```
PROC configurable screen clock :
   delay := 0 ;
   do
      do
         show (time + delay) ;
         timeout := 60s − time mod 60s ;
         receive (client, msg, timeout)
      until msg received od ;
      delay := msg.time shift
   od
END PROC
```

This example also shows that Atkinson's original scale of persistence [Atk 83] (transient, local, own/global, ...outliving the program) must be widened by a second dimension, the lifetime of the process. Since the process itself is persistent, the variable 'delay' lives forever, even though it is local to the procedure.

Tasks maintain the data objects and exclusively control access to them, and, in consequence, there are no data objects outside a task. (If a data object is not owned by at least one task, it no longer exists.) Therefore, directories, file servers and databases are most naturally implemented in Eumel and L3 as persistent tasks containing persistent data and persistent threads.

Tasks and threads are first class objects: they

- have unique identifiers, which are even unique over time (to date, 64 bits are used per task or thread id),
- are active, autonomous and (in most cases) persistent,
- communicate via secure channels.

Thus global naming and binding problems at the μ-kernel level are reduced, applying only to the field of inter-process communication. The integrity of message transfer, the uniqueness of task and thread identifiers and one distinguished identifier are sufficient to solve these problems. The last item can be used as a root for higher level naming schemes.

Since all data objects are completely controlled by tasks, local identifiers are sufficient, unique only per task and not in time; this simplifies the kernel. Furthermore higher levels gain flexibility, because they are less restricted by kernel concepts. Some naming and binding mechanisms that have been implemented on top of the kernel are described in section 5.

4 Details

This section gives a short overview of some details of the support for persistence in Eumel and L3. More details are described in [EUM 79, EUM 79a, Bey 89, Lie 91, Lie 92].

4.1 Tasks, threads and communication

The L3-kernel is an abstract machine implementing the data type *task*. A task consists of

- at least one *thread*
  A thread (like a Mach thread) is a running program; up to 16384 are allowed per machine. All threads, except resident (unpaged) driver or kernel threads, are persistent objects.
- up to 16383 *dataspace*s
  A dataspace is a virtual memory object of size up to 1 GB. Dataspaces are also persistent objects and are subject to demand paging. Copying and sending is done lazily. Physical copy operations are delayed (copy on write) as in Accent, Mach and BirliX.
- one address space
  Dataspaces are mapped dynamically into the address space for reading or modification. For hardware driver tasks, the address space is logically extended by the IO ports assigned to the task. (Recall that all device drivers are located outside the kernel and run at user level.)

As in Mach, paging is done by the default or external pager tasks, hence all interactions between tasks, and with the outer world, are based on inter-process communication (ipc).

The Eumel/L3 ipc model is quite straightforward. Active components, i.e. threads, communicate via messages which consist of strings and/or dataspaces. Each message is sent directly from the sending to the receiving thread. There are neither communication channels nor links, only global thread and task identifiers (ids).[1] L3 ipc therefore involves neither concepts of opening and closing nor additional kernel objects such as links or ports. This simple model differs in two important respects from most other message oriented systems:

- absence of explicit binding (opening a channel)
  One would expect additional costs, because the kernel must check the communication's validity each time a message is sent, but, on the other hand, there is no need for bookkeeping of open channels. We consider that our approach is more elegant, and, in fact, communication in L3 is fast (see below).

- no message buffering
  Due to the absence of channel objects we have synchronous ipc only; sender and receiver must have a rendezvous. But practice has shown that higher level communication objects, such as pipes, ports and queues, can be implemented flexibly and efficiently on top of the kernel by means of threads.

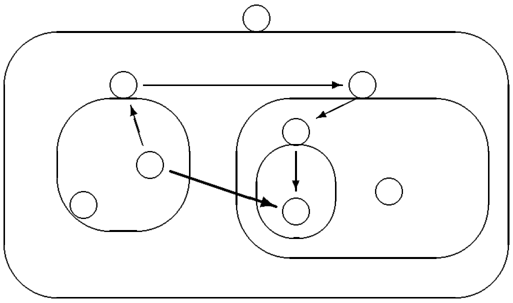
As described in [Lie 93], L3's ipc implementation performs well: short message transfer is 22 times faster than Mach and 5 times faster than QNX [Hil 92]. This allows to integrate even hardware interrupts as ipc.

Local ipc is handled by the kernel, remote ipc by user level tasks, more precisely by chiefs (see section 4.2). Tasks and threads have unique identifiers, unique even in time, so a server usually concludes from the id of the message sender whether the requested action is permitted for this client or not. The integrity of messages (no modification, no loss, no sender id faking), in conjunction with the autonomy of tasks, is the basis for higher level protection.

4.2 Clans & Chiefs

Within both the L3 system and other systems based on direct message transfer, e.g. BirliX, protection is essentially a matter of message control. Using access control lists (acl) this can be done at the server level, but maintenance of large distributed acls becomes hard when access rights change rapidly. So Kowalski and Härtig [Kow 90] have proposed that object (passive entity) protection be complemented by subject (active entity) restrictions. In this approach the kernel is able to restrict the outgoing messages of a task (the subject) by means of a list of permitted receivers.

The clan concept [Lie 92], which is unique to L3, is an algorithmic generalization of this idea:



A *clan* (denoted as an oval) is a set of tasks (denoted as a circle) headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (including the sender and receiver ids as well as the contents) and decide whether or not it should be passed to the destination to which it was addressed. As demonstrated in the figure above, these rules apply to nested clans as well. Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control. Typical clan structures are

Clan per machine: In this simple model there is only one clan per machine covering all tasks. Local communication is handled directly by the kernel without incorporating a chief, whereas cross machine communication involves the chief of the sending and the receiving machine. Hence, the clan concept is used for implementing remote ipc by user level tasks.

Clan per system version: Sometimes chiefs are used for adapting different versions. The servers of the old or new versions are encapsulated by a clan so that its chief can translate the messages.

Clan per user: Surrounding the tasks of each user or user group by a clan is a typical method when

---

[1] How does one acquire the id of a new partner for communication? This is achieved using one or more name servers. Usually the creating task implants the id of at least one name server into a newly created task. Using these or the root id the new task can communicate with some name server to get further task/thread ids.

building security systems. Then the chiefs are used to control and enforce the requested security policy.

Clan per task: In the extreme case there are single tasks each controlled by a specific chief. For example these one-task-clans are used for debugging and supervising suspicious programs.

In the case of intra-clan communication (no chief involved), the additional costs of the clan concept are negligible (below 1% of minimal ipc time). Inter-clan communication however multiplies the ipc operations by the number of chiefs involved. This can be tolerated, since (i) L3 ipc is very fast (see above) and (ii) crossing clan boundaries occurs seldom enough in practice. Note that many security policies can be implemented simply by checking the client id in the server and do not need clans.

4.3 Dataspaces and pagers

Data objects can be mapped into address *spaces* and then arbitrarily addressed, therefore they are called *dataspaces*. They are managed by either the default or their individually associated external pagers. The default pager implementing persistent data objects was one of the key components of Eumel and L3 from the very beginning (1979), whereas Mach's external pager concept was integrated later as a generalization. External pagers are user level tasks. They may use devices (different from the default swapping device) for paging as well as virtual objects which are managed by other pagers, i.e. pagers may be stacked.

Dataspaces can be created, deleted, mapped, unmapped, copied and sent as parts of messages to other tasks. Copying and sending are done lazily. Since copy on write is supported by the kernel's memory management and the default pager, even file copies are cheap. Lazy copying and lazy sending use techniques similar to shadow paging [Lor 77] but are totally symmetric. When modifying the original of a file the corresponding page must be physically copied (or mapped to a new backing store block) in the same way as when modifying the copy.

4.4 Fixed points

Since processes are as persistent as data objects, checkpoint/recover mechanisms covering both are needed: the most important one in Eumel and L3 is the *fixed point*, which is also denoted by the slang word *fixpoint*. A fixpoint is a checkpoint covering the complete system (of one node). It includes a *consistent* copy of all tasks, threads, address spaces and dataspaces (i.e. all processes, files, databases, terminal screens etc.) taken at *the same point of time*. Due to copy on write techniques, this atomic action is short enough not to disturb the users. Writing the dirty pages to the backing store is done afterwards, concurrently to normal user and system activity.

To implement the fixpoint, all dataspaces of all tasks and all thread and task control blocks are integrated into the 'dataspace of all dataspaces' (dsds). So lazy copying can be applied to the system as a whole. Using a slightly modified ELAN notation, the fixpoint algorithm is:

```
begin atomic
   new fixpoint dsds := actual dsds
end atomic ;
for i from 0 upto max frame number do
   if frame[i].dirty ∧ frame[i].belongs to fixpoint
      then write to backing store (frame[i])
   fi
od ;
begin atomic
   delete old fixpoint
   make new fixpoint valid on backing store
end atomic .
```

The first atomic action, copying the dataspace of all dataspaces, requires a physical copy of the system root pointer (32 bytes) and flushing of all page frames which are mapped with write permission in this moment. On a 25 MHz Intel 386-based machine with 12 MB main memory this costs between 2 and 5 milliseconds. The following cleanup phase runs concurrent to the normal user and system activities at a relatively low priority. On average, one third of main memory contains dirty frames; depending to a great degree on the actual workload, between 7 and 150 seconds are needed for complete write back. The final atomic action writes one sector to the backing store, this takes about 15 milliseconds.

Since all active and passive objects of one machine (including the terminal screens) are frozen consistently at the same time, fixpointing and recovery is transparent for most applications. Clients and servers on the same machine are always in a consistent state.

However, this mechanism is not sufficient for a distributed system, since a "worldwide" fixpoint is not reasonable, especially not in a heterogenous distributed system. Therefore explicit (and sometimes application specific) recovery policies are necessary whenever relevant partners are not included in the fixpoint. Since the μ-kernel itself is not distributed, these policies must be implemented on top of it. Usually, general policies are based on the *clan*-mechanism.

Fixpoints occur periodically or user initiated, but their latency limits their frequency. A practical lower bound is one fixpoint all 3 minutes. A common method for implementing more fine-granular checkpoints is to use an external pager which supports checkpointing each dataspace separately. By this, a remote file server can also save each modified file independently of the various fixpoints in the distributed system. But note that even such servers consist of persistent tasks and threads which are fixpointed by the default pager.

A regular shutdown consists of enforcing a fixpoint and stopping all other threads. There is no relevant difference to an irregular breakdown. In both cases restarting the system is done by activating the last valid fixpoint.

4.5 Fixpointing the kernel

Recovering a task or restarting a thread needs not only user level data (stacks, dataspaces) but also thread state, priority, mapping information, saved processor registers etc. All these kernel data is held in thread and task control blocks which are *virtual* objects. They are located in a kernel dataspace being subject to paging, and the ordinary fixpoint mechanism checkpoints them like all other objects. Thus the control blocks are as persistent as user data, and recovery is simple as far as only values and virtual (kernel and user) addresses are involved.

For illustration let us assume that a message transfer running between two persistent threads is interrupted by a page fault (or time slice exhaustion or external interrupt) and that this situation is frozen by a fixpoint. When recovering the system from this fixpoint, inspection of the thread states (part of the persistent thread control blocks) leads to restarting the message transfer. Recall that the data and the control block addresses are virtual so that changes of real memory allocation are transparent to ipc. While fixpoint/restart is transparent to ipc between persistent threads (even to timeout handling), communication with a transient thread (e.g. a resident device driver) is aborted on restart.

In fact, holding the control blocks as virtual objects does not solve all problems related to thread restarts: Although kernel stacks are part of the control blocks and therefore persistent, and stack addresses are virtual addresses, simply loading the stackpointer does not work, since the stacks may contain real addresses which necessarily come up when handling page faults. Furthermore, changing the kernel code would invalidate return addresses.

Therefore the restart algorithm inspects the kernel stack bottom of each thread and decides how to restart it:

Waiting for ipc: The thread remains waiting. Its state is not changed but the return addresses on the kernel stack are updated, since the kernel code may have changed.

Running ipc: The thread state is not changed, but its kernel stack is reset to the outmost level and its instruction pointer set to a special ipc restart routine.

Otherwise: The thread state is set to 'busy' and restarted at user level. (This is possible, since all system calls behave atomically, i.e. have no effect until completion.)

5 Applications

5.1 Programs and procedures

One consequence of task persistence is that there is no real difference between running a program and calling a procedure within a program. Conventionally, running a program means:

```
allocate space ;
copy program code from load file into memory ;
relocate code ;
call (start address) .
```

The first three actions are not required in a persistent task. If the compiler takes advantage of this feature (as Eumel's ELAN compiler does), it simply generates executable code at the appropriate virtual addresses. Then running a program is

```
call (start address) .
```

By means of this mechanism, and by using lazy copying for its table initialization, the ELAN compiler achieves noteworthy speed, especially when translating small programs. For example, to translate and execute '`put ("hello world")`' costs 48 ms total elapsed time on a 25 MHz 386 based machine, 10.6 ms on a 50 MHz 486. This speed permits the translation of each job control and editor command as a separate single program.

If programs with conflicting addresses should subsequently run in the same address space, and the executable code of each program is contained in a file, a program is executed by:

```
if required region of address space already used
   then unmap corresponding dataspaces
fi ;
map file to required region of address space ;
call (start address) .
```

5.2 Dataspaces and persistence

Dataspaces can be mapped into an address space and may contain data of all types, i.e. Atkinson's type completeness requirement [Atk 83] is fulfilled. Variables located in mapped dataspaces are accessed by the same mechanism (by virtual address) and with the same efficiency than normal program variables. In fact, programs are contained in dataspaces as well.

Note that persistence here does not require open/close operations. Due to task persistence a dataspace may remain mapped arbitrarily long. Distributed applications use external pagers and remote procedure calls (RPC) for synchronization and/or remote object access.

5.3 Files, directories and protection

By default, directories and file server are implemented as tasks. These use simple arrays to hold the catalogue (file names and attributes) and dataspaces as files:

```
initialize catalogue as empty ;
do {forever}
   wait for order ;
   if from authorised partner $\wedge$ access permitted
      then if is file access order AND file exists
               then send or map dataspace
           elif is create order $\wedge$ $\neg$ file exists
               then allocate catalogue entry;
                    create dataspace
           elif ...
           fi
      else reject
   fi
od .
```

Here 'access permitted' can be an arbitrary algorithm usable to support various security policies, e.g. 'free access', simple 'password protection', 'time bound access', 'authorized user only access' etc. The mechanism permits optional as well as mandatory access control.

6 Experiences

6.1 Stabilities and instabilities

We strongly recommend that system developers use their own system as a workbench (not only for testing) as early as possible. Only if they take the risk of their own software instabilities and inefficiencies, will the system soon become stable and fast enough for end users. In particular, relying on the system will lead to a realistic balance between stability and optimization.

When we began to use Eumel as a workbench, we were particularly afraid of collapsing processes. Recall that the shutdown of a task makes all of its data, i.e. all of its local files, inaccessable. Of course, we could have held all files in a file server task, so that they would not be seriously affected by a collapsing worktask. However, although the probability of shutdown in a file server task may be lower than in a task used for editing, compiling and testing new programs, the file server itself is also a task, potentially suffering from the same threats. In fact, the very first Eumel versions we (and only we) used, led to disasters like:

- subscript overflows in directory handlers resulting in broken chains and unusable directories,
- stack overflows within the stack overflow exception handler resulting in absolutely dead tasks,
- crashes of the virtual machine when executing instructions under special circumstances.

However, most of these crashes turned out to be soft, i.e. recoverable. We cured them by simply resetting the system to the last fixpoint. On average the previous 5 minutes were lost, but serious harm had been avoided and we could correct our software.

The fixpoint mechanism made the system more robust, not only in its early stages but also when it became widely used for production. The majority of faults survived by means of fixpoints, thereafter, were the originally intended hardware related ones, e.g. power failure, controller hangups etc. and spurious kernel faults.

> The most prominent use of the fixpoint facility occurred when we gave a demonstration for the German Minister of Research and Technology. Shortly before the minister entered the room, a cleaner hurried in to remove the last speck of dust. Of course, her first action was to unplug our power supply cable and plug in her vacuum cleaner. It provided a perfect demonstration of fault tolerance in a persistent system by fixpoint.

Besides such positive effects of stability and robustness through fixpointing, we also had bad experiences. Unfortunately the fixpoint stabilises a corrupted system in the same way. In practice, this only happened in conjunction with kernel level errors. The overwhelming majority of severe kernel errors led to an immediate crash or starvation and did not hurt the fixpointed system, but some errors in the areas of memory management, paging and concurrent block garbage collection led to serious and permanent low level defects. In most cases synchronization errors resulted in a multiply used block on disk: usually one instance stored a page block table in a disk block, while a different one used the same block for a text file.

In the first years of Eumel these errors soon became very rare, but three unpleasant effects became noticeable:

- Severe kernel errors tended to be extreme rare and only to occur at one or two installations (far away, of course), and not correlated with specific actions.
- About one third of these problems turned out to be spurious hardware malfunctions, which in most cases were hard to detect and to prove.
- Improving the kernel algorithms and incorporating new hardware (memory bank switching, specific CPU features etc.) was required by the users, but often led to new periods of higher instability.

By way of illustration, here is one example incident:

> There was a machine tool manufacturing plant using L3, which had on average (but not periodically) one system breakdown every two weeks. The crashes often occurred early in the morning, when nobody was working and there were no special system activities. Analyzing the crashes did not give usable hints to us. At that time about 200 other L3 systems were in use, and none of them suffered from kernel instability. We inspected the special application software running on the critical system, but we did not find anything extraordinary. For two months we racked our brains performing "dry" analysis of potentially critical situations. Indeed, we found two errors which certainly had never occurred, but not the critical one. Then the installation was upgraded from 8 MB to 12 MB main memory, and the problem immediately disappeared.
>
> But: When simplifying the kernel algorithms in the following year, we discovered three errors, each of them being very very improbable but able to explain the spurious crashes. Since the machine tool manufacturer did not want to return to his potentially unstable hardware to allow further tests, we will never know for certain whether the crashes were induced by those errors. Intuitively I would give the odds 6 : 1 : 3 for 'fixed L3 bug' : 'as yet unknown L3 bug' : 'hardware malfunction'.

Concluding the good and bad experiences of 13 years as far as stability in a persistent system is concerned:

- A persistent system can be made very stable. With Eumel and L3 we have reached a state where crashes (recoverable and unrecoverable) due to software are less frequent than those due to hardware.
- The overwhelming majority of crashes are recoverable.
- Even when the system is as stable as possible, if you wait long enough there will still suddenly be someone suffering from a really damaged system.

6.2 Performance

Frequently, people believe that persistence is expensive, especially when taken together with fault tolerance. Due to our experiences with Eumel and L3, we are convinced that it is relatively cheap, if the system is carefully and thoroughly designed, from hardware up to application level.

Unfortunately we cannot prove that persistence is inexpensive, since there is no nonpersistent Eumel or L3, but we are able to give two arguments that support our claim:

- Apart from fixpointing, there is no feature that is required purely in order to provide persistence. Virtual address spaces and paging are common features of timesharing systems.
- There are commercial 16–user–systems running L3 on a single 486 monoprocessor.

Fixpointing, however, is not for free. It costs a certain amount of backing store, additional swap outs for storing the fixpoint and some free main memory for copy on write actions taking place immediately after starting the fixpoint and before its frames are written back to disk.

Experience has shown that for most applications these additional costs become negligible when 30% more main memory — reducing normal paging circumvents the disk traffic bottleneck during fixpoint — and a disk reserve of 3 times main memory size are available.

Although measurements of a few special and singular mechanisms are not adequate when seriously discussing a system's performance, some are presented here simply to give the reader a rough feeling. A 50 MHz 486 noname PC with 16 MB main memory was used for these measurements:

null cross address space RPC (user ↔ user)	10 μs
copying 300 files to another task (including directory updates)	2.6 ms/file
create dataspace; map it; write one byte; delete dataspace	2.0 ms
fixpoint covering 1000 dirty page frames (no concurrent activity)	6.8 ms/page
translate and execute 'put ("hello world")'	10.6 ms

6.3 How programmers adopt the ideas

We have used Eumel to teach word processing to university secretaries who had no previous computing experience whatsoever. were completely unexperienced concerning computers. We soon learned that we must not talk about persistence: they took persistence for granted. It was a natural concept for them (paper is persistent), but our complicated explanations of persistence irritated them. When giving the next course to secretaries we therefore reduced our explanation of persistence to: "For obscure technical reasons an unexpected power failure will destroy the work you have done in the preceding few minutes."

Pupils or students learning programming using ELAN over Eumel/L3, at first look at files as something which can be edited. They regard persistence as a natural but secondary effect. Later on, they write modules which are naturally based on the persistence of variables, e.g.:

```
test var actual password := "" ; {static variable}

proc set password (text old, new) :
   if old = actual password
      then actual password := new
   fi
end proc set password ;


bool proc password ok (text pw) :
   (pw = actual password)
end proc password ok ;
```

Although this type of application is produced as a matter of course, students start to think explicitly about persistence when they first write programs for long running computation. For a beginner, the persistence of a 'really executing' activity seems to be less natural than that of a 'waiting' activity.

Things are different when experienced software engineers first use Eumel/L3. They know about long running jobs from their experience of mainframes, but they tend to hold as much data as possible in files. They soon start to map files/dataspaces holding arbitrary data structures, but only construct applications, based on a multitude of persistent tasks, after they have gained considerable experience. Often they structure their system on tasks to separate responsibilties. For example, the lawyer supporting system has about 10 tasks per user. Sometimes this leads to one task per developer in the same way that three compiler writers working together tend to construct a three pass compiler.

An unintended way using persistence is practiced by some experienced users: having deleted the wrong file (despite the system asked back), pressing reset resets the system to the last fixpoint and the deleted file reappears. Usually, colleagues who are using the same machine concurrently prohibit them from doing this.

7 Conclusions

Persistence as a basic and universal principle covering data and processes has proved to be a good foundation for Eumel and L3. Our experiences of the last 13 years have shown that programmers like persistence, and both system level and application programming become easier. We have learned that realizing persistence at the μ-kernel level permits the construction of an efficient and flexible system with a reasonable amount of work for design and implementation. However, a system relying heavily on persistence needs significant effort to make it stable enough for real use.

Acknowledgements

The development of Eumel and L3 would have been impossible without the contributions and programming efforts of Uwe Beyer, Dietmar Heinrichs and Peter Heyderhoff. I would also like to thank Peter Dickman for proofreading this paper and helpful comments. This paper was written using LaTeX on L3.

References

[Acc 86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. *Mach: A New Kernel Foundation for UNIX Development.* Proceedings Usenix Summer'86 Conference. Atlanta, Georgia, June 1986, pp. 93-113.

[Alb 80] A. Albano, M. E. Occuchiuto, R. Orsini. *A uniform management of temporary and persistent complex data in high level languages.* Nota Scientifica, S-80-15, September 1980.

[Atk 82] M. P. Atkinson, K. J. Chisholm, W. P. Cockshott. *PS-Algol: An Algol with a persistent heap.* Sigplan Notices 17(7), July 1982, pp. 24-31.

[Atk 83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, R. Morrison. *An Approach to Persistent Programming.* Computer Journal 26(4), November 1983, pp. 360-365.

[Atk 87] M. P. Atkinson, O. P. Buneman. *Types and Persistence in Database Programming Languages.* Computing Surveys 19(2), June 1987, pp. 105-190.

[Ben 72] A. Bensoussan, C.T. Klingen, R.C. Daley. *The Multics Virtual Memory: Concept and Design.* CACM,15, 5, May 1972, pp. 308–318.

[Bey 89] U. Beyer, D. Heinrichs, J. Liedtke. *Dataspaces in L3.* Proceedings MIMI '88, Barcelona, July 1988.

[Cah 93] V. Cahill, R. Balter, N. Harris and Rousset de Pina (Eds.). *The Comandos Distributed Application Platform.* Springer-Verlag 1993 (to appear)

[Che 79] D. Cheriton, D. A. Malcolm, L. S. Melen, G. R. Sager. *Thoth, a Portable Real-Time Operating System.* CACM 22(2), February 1979, pp. 105-115.

[Che 84] D. Cheriton. *The V Kernel: A Software Base for Distributed Systems.* IEEE Software, pp. 19-42, April 1984.

[EUM 79] *EUMEL Benutzerhandbuch* (German). University of Bielefeld 1979.
EUMEL User Manual (English). GMD 1985.

[EUM 79a] *EUMEL Referenzhandbuch* (German). University of Bielefeld 1979.
EUMEL Reference Manual (English). GMD 1985.

[Gui 82] M. Guillemont. *The Chorus Distributed Operating System: Design and Implementation.* Proceedings ACM International Symposium on Local Computer Networks, Firenze, April 1982.

[Här 92] H. Härtig, W.E. Kühnhauser, W. Reck. *Operating Systems on Top of Persistent Object Systems –The BirliX Approach –.* Proceedings 25th Hawaii International Conference on Systems Sciences, IEEE Press 1992, Vol 1, pp. 790–799.

[Har 85] N. Hardy. *The Keykos Architecture.* Operating Systems Review, September 1985.

[Hil 92] D. Hildebrand. *An Architectural Overview of QNX.* Proceeedings Micro-kernel and Other Kernel Architectures Usenix Workshop, Seattle, April 1992, pp. 113-126.

[Hom 79] G. Hommel, J. Jäckel, S. Jähnichen, K. Kleine, C.H.A. Koster. *ELAN Sprachbeschreibung.* Wiesbaden 1979.

[Kow 90] O. Kowalski, H. Härtig. *Protection in the BirliX Operating System.* Proceedings 10th International Conference on Distributed Computing Systems, 1990.

[Lie 91] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, G. Szalay. *Two Years of Experience with a μ-Kernel Based OS.* Operating Systems Review, 2, 1991.

[Lie 92] J. Liedtke. *Clans & Chiefs.* Proceedings 12. GI/ITG-Fachtagung Architektur von Rechensystemen, Kiel 1992, A. Jammel (Ed.), Springer-Verlag

[Lie 92a] J.Liedtke. *Fast Thread Management and Communication Without Continuations.* Proceeedings Micro-kernel and Other Kernel Architectures Usenix Workshop, Seattle 1992.

[Lie 93] J.Liedtke. *Improving IPC by Kernel Design.* Proceedings 14th ACM Symposium on Operating Principles, Asheville, North Carolina, December 1993.

[Lor 77] R.A. Lorie. *Physical Integrity in a Large Segmented Database.* ACM Transactions on Database Systems, 2, 1, March 1977, pp. 91–104.

[Mul 84] S.J. Mullender et al. *The Amoeba Distributed Operating System: Selected Papers 1984–1987.* CWI Tract. No. 41, Amsterdam 1987.

[Ras 81] R. Rashid, G. Robertson. *Accent: A Communication Oriented Network Operating System Kernel.* Proceedings 8th Symposium on Operating System Principles, December 1981.

[Ros 87] J.L Rosenberg and J.L. Keedy. *Object Management and Addressing in the MONADS Architecture.* Proceedings 2nd International Workshop on Persistent Object Systems, Appin 1987, available as PRRR–44, Universities of Glasgow and St. Andrews.