Gracle
A development and deployment environment for Common Lisp

Robert Strandh

December 23, 2004

1 Introduction

This document describes what we call a "development and deployment environment" that we would like to implement. Such an environment constitutes the part of an operating system that is visible to application programs.

To avoid having to implement device drivers and other low-level code, we are planning to implement this environment on top of Linux, as an ordinary Linux process.

Gracle differs in two important ways from ordinary operating systems such as GNU/Linux, in that it does not have files and that it does not have processes. The next two sections elaborate on this fact.

1.1 Gracle does not have files

Gracle does not have files in the traditional meaning of files. It does not make a distinction between primary and secondary storage. The semantics that apply to all Gracle object is that of random-access memory. Files are replaced by strings and other structured objects containing characters. This implies that Gracle has a *single-level store*.

Other operating systems have had single-level stores. Multics did in the 1970s. The problem with Multics was that it was not crash proof (luckily, it rarely crashed). More recently EROS (Extremely Reliable Operating System) is an operating system with a crash-proof single-level store. We are planning to use the EROS model for Gracle, except that we do not plan to use the naked disk and instead implement the permanent store as a Linux file.

The great advantage of not having files in traditional sense is that applications do not have to convert its objects to sequences of bytes in order for data to persist between invocations. Structured objects in main memory are all persistent.

1.2 Gracle does not have processes

By *process*, we mean a thread of execution with its own *address space*. Processes are useful in a couple of ways.

First, they allow for several programs to run on a computer with a relatively small address space. Without separate address spaces in each process, this relatively small address space would have to be divided up between all currently executing programs. Now that computers with 64-bit pointers are becoming common, for the first time, this reason for using processes is no longer valid.

Second, it allows for an application program to run arbitrary code without running the risk of modifying data of other applications. This is a great advantage, and by abandoning processes in Gracle, we have to introduce some other means of protection between application programs. In this document, we shall not discuss this particular aspect of Gracle (simply because we haven't figured it out yet).

In Gracle, all threads of execution are *light-weight processes*, commonly known as *threads*. There are huge advantages of sharing the address space between all threads of execution. For one thing, an application does not have to convert objects to byte streams in order to give an object to a different application. Instead, it can just hand a pointer to a structured object in main memory. Also, copying objects becomes necessary much less often.

1.3 Objects are not organized in a hierarchy

This section is very preliminary, as we have not worked out the details yet.

We would like to eliminate the hierarchy of objects (files) that has become standard since it was introduced in the 1960s. Most often, the order of the directories in the path of an object in such a hierarchy is not meaningful to the user, but imposed by the hierarchy.

Also, we would like to be able to construct collections of object on the fly, and not be limited to the collections that the directories of a hierarchy imposes, for instance "the collection of all email messages less than a month old and with a PDF attachment".

While we could settle with the means of access to objects provided by Common Lisp (special variables), it seems useful to have some kind of "data base"

of objects that are accessible by queries like the one above. Obviously, we do not want every cons cell to be accessible this way. For that reason, we distinguish between objects that are *archived* and objects that are not. The concept of archived objects is different from that of *persistent objects*. In Gracle, all accessible objects are persistent in that they do not disappear at logout our system shutdown. Archiving an object just means giving it certain properties and making it accessible by queries like the one above. Such properties include creation date and perhaps a number of *tags* that serve the same purpose as directory names in traditional hierarchical systems. Tags and dates are not stored in the objects themselves, but in data structures (such as hash tables) external to the objects. This way, it is possible to archive very small objects such as cons cells without it being necessary to reserve space in each cons cell for archive information.

In a traditional hierarchy, the *current directory* serves two different purposes. The first is to serve as a current collection of objects that is currently manipulated by the user. This collection is determined by the fact that all objects in the collection share the same prefix in the path name. The second purpose is to give newly created objects this prefix so that such objects become members of the same collection.

For Gracle, We suggest using a set of *current properties* defining the current collection of objects, and a set of *assigned properties* that define what properties newly archived objects will inherit. They are not the same, since the current properties might include restrictions on date of creation and any other arbitrary filter, whereas newly archived objects would be assigned some properties automatically (date) and others would come from the assigned properties (such as whether the object is part of some named collection of documents).

2 How Linux memory management works

Memory management is done in chunks of typically 4kbytes called *pages*. We shall assume that there is a single chunk size for both primary and secondary (disk) memory. That is not necessarily true, but it does not matter for the discussion here. The *page number* is simply the address of a page shifted right 12 position ($2^{12} = 4k$).

We distinguish between *virtual addresses*, *physical addresses*, and *disk addresses*.

A virtual address is one that is seen by application code. In a typical 32-bit Linux system, virtual addresses go from 0 to $2^{32} - 1$. In a 64-bit system, they would go from 0 to $2^{64} - 1$. Since there is usually less physical memory that virtual addresses, the virtual address space contains holes that have no physical memory associated with them. Any attempt for application code to address such space results in a signal being sent to the application.

The memory management unit (MMU) of the processor translates from virtual to physical addresses. This translation is done at each memory access and is therefore very fast (thanks to something called a translation lookaside buffer (TLB)). In this document, we shall not discuss physical addresses very much, since they are of no interest to the rest of the system. In Linux, the MMU translation is altered at context-switch time (when a process is taken off the processor and another one is put on). In Gracle, since we do not have processes, we do not have to alter the MMU mapping at context switches (we may have to alter protections, though).

Linux, has what is traditionally called *virtual memory*, i.e., a process can have more valid addresses in its virtual address space than there is main memory in the system. When the process touches a part of its virtual address space that is not in primary memory at the time, the system will evict an existing page to a part of the disk traditionally called the *swap space*, for historical reasons. The page being touched is then read in from the swap space to the page in primary memory that was evicted, and the MMU is altered to reflect the new position of the primary memory page in the virtual address space. This process is called *paging*. In fact, paging is required as soon as the sum of the virtual space needed by each process is greater than the size of primary memory, which is fairly common. On a modern personal computer, it is rare that a single process needs more memory than there is primary memory (especially since the size of primary memory is rapidly approaching the size of the address space in a 32-bit system). For that reason, paging is mainly necessary after a context switch.

For each process, the system keeps a *page table* that maps virtual page numbers to page numbers on disk. When a page is evicted, the page table is consulted to determine where on disk it is to be written, and when a new page needs to be read in, the page table is again consulted to determine where on disk the new page is to be found. Page tables can be quite large, since they potentially need to map every page in the virtual address space to a page on disk. In a 32-bit system, each process can have 2^{20} (roughly a million) pages. Storing the page tables for process might itself require 1000 pages. For that reason, page tables are often stored in a tree where nodes are allocated only if the page is allocated to the process.

Since we are planning to use a Linux file to implement the backing store of Gracle, we need to know a little about such files. The `open` system call takes a character string (the name of a file) and returns a small integer known as the *file descriptor*, which is really just an index into a per-process table kept by the operating system. The `read` system call takes a file descriptor, a pointer to a byte vector and an integer indicating the number of bytes to read. It transfers the bytes from the disk file to the vector in the virtual memory of the process. The `write` system call does the inverse. Since a file can partly reside in a disk cache (which consists of volatile semiconductor memory) there is a system call `fsync` that guarantees that data is written to disk.

For the purpose of this document, there are only a few more Linux system calls that interest us.

The `mmap` system call modifies the page table by associating parts of the virtual address space with pages on disk, either in a file, or in the swap space. It can also remove pages and thus make a hole in the virtual address space, and it can be told to use only primary memory for a page in the virtual space (and no backing store).

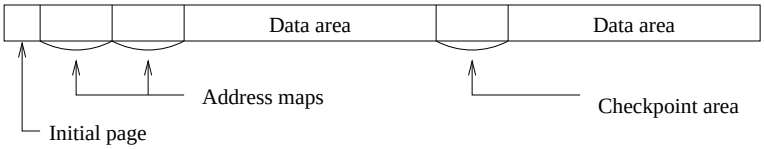
The `mprotect` system call alters the protection of pages in virtual memory. The `msync` system call is similar to the `fsync` call, but is used on memory that has been mapped by `mmap`.

3 How the EROS single-level store works

This section contains a description of my understanding of how the EROS single-level store works.

3.1 Disk layout

The disk (we will assume there is only one) has the following layout:



The initial page contains various information that is required by the system at boot time. In particular, it contains a bit that indicates whether the first or the second address map is to be used.

Virtual addresses are actual addresses on the disk, except that a few gigabytes (a few million pages, if we assume a page is 4kbytes) might be located elsewhere (in the checkpoint area) on the disk. These exceptions are recorded in one of the following address maps. that are organized as hash tables mapping virtual page numbers to physical addresses on the disk. The address map indicated in the initial page contains references to pages in the checkpoint area.

The two data areas are where ordinary objects are allocated.

The checkpoint area is allocated in the middle of the disk so as to minimize seek times for pages in this area. The space allocator will not allocate space in the checkpoint area.

3.2 Terminology

We distinguish between three different types of pages: *virgin* pages that were paged in from one of the data areas and never modified, *clean* pages which were modified at some point, but then paged out to the checkpoint area and not modified after that, and *dirty* pages that have space assigned to them in the checkpoint area, but it is the in-core version that is up to date. Virgin pages are always write protected. Other pages are sometimes write protected sometimes not.

3.3 Normal operation

The system manages a checkpoint area on disk that is viewed as a circular buffer containing 4 sequential consecutive zones, the *primary* zone, the *secondary* zone, the *tertiary* zone, and the *free* zone. The system is free to read from and to write to the primary and the secondary zone in that they do not occur in the current address map on disk, so that writing to them does not affect the consistency of the disk version of the system. The tertiary zone contains exactly those pages that occur in the current address map on disk, so they cannot be written to until the current address map on disk has been changed to exclude those pages.

Corresponding to the primary zone, there is a in-core primary address map that maps virtual page numbers to pages in the primary zone. Similarly, corresponding to the secondary zone, there is an in-core secondary address map that maps virtual page numbers to pages in the secondary zone. These maps are organized as hash tables.

The system can be in two different states: the *save* state and the *migrate* state according to the activity in the secondary zone. In the save state, the main activity of this zone is to write dirty pages to disk. In the migrate state, pages in the zone are migrated to their home locations in the data area.

Presumably, main memory is fully used during normal operation. When a page fault occurs as a result of mutator execution, some page must be selected to expire. An ordinary LRU-style algorithm can be used. No write operation is needed if the expired page is virgin or clean. Now, we must find the location on disk of the requested page. If that page is in the primary address map, the page is in the primary zone. In that case it is loaded in as usual and execution continues. If it is not in the primary address map, but in the secondary address map, it is loaded from the secondary zone, and write protected. If it is neither in the primary address map nor in the secondary address map, it is loaded from its home location and write protected.

When a write fault occurs (we assume that the page fault has already occurred so that the page is in-core), we allocate space for the new version of the page

in the primary zone (by moving the first page in the free zone to the primary zone) and update the primary address map to reflect this allocation. If the page is a dirty page (which must then be in the secondary zone), we write it to disk in its location in the secondary zone, and update the secondary address map to reflect that the page is now clean. Nothing is written to the primary zone. The page is unprotected.

A checkpointing process runs in parallel with mutator execution. When the system is in the save state, this process scans the secondary address map for dirty pages and systematically saves them to the secondary zone. When no more such dirty pages exist, the secondary address map is written to the unused address map area on disk and the initial page is finally written to reflect the change in on-disk address map. Now the tertiary zone in the checkpoint area is appended to the free zone, and the system changes state to migration.

In the migrate state, the checkpointing process scans the secondary address map. For each page it finds, it checks whether the page also occurs in the primary address map. If that is the case, the entry is simply deleted from the secondary address map, since the next checkpoint on disk will indicate that this page is in the primary zone. If the page is not in the primary address map, it is migrated to its home location, and its entry in the secondary address map is removed. When the secondary address map is empty, the system is ready for a new checkpoint.

To declare a new checkpoint, all pages that have entries in the primary address map are write protected. The secondary zone becomes tertiary, the primary address map becomes the secondary address map, a new, empty, primary address map is created, and the system is put into save state.

3.4 System boot

When the system boots, frames (pages in main memory) are allocated for the initial page and the two address maps. The rest of the address space is unassigned (has no frames allocated to it). The initial page is read in, as well as the address map indicated to be in use by the first page. Presumably, the first page also contains initial processor information such as register contents, etc. which will allow execution to start.

The address map that is read in from disk becomes the secondary address map. The primary address map is initialized to be empty. The secondary zone is initialized to the pages that are mentioned in the secondary address map. The primary and tertiary zones are initially empty.

Since the address space containing actual application objects is not yet assigned, the system will soon take a page fault. This will result in the allocation of a frame to a page of the address space, and in the reading of the page from disk to

the frame. Many such page faults will occur in the beginning until the working set of the system has been loaded into main memory.

4 Implementing the single-level store

In the previous section, we described how the EROS single-level store works. In this section, we adapt that description to an implementation on top of Linux, or perhaps on any Posix-compliant system.

The EROS disk will be simulated with a Linux file. It would be great if we could influence the placement of the disk blocks of the file so as to obtain the performance that EROS is capable of, but even if that is not possible, we can still get the semantics that we want.

The logical layout of the file would be the same as that of the EROS disk described in the previous section.

We imagine that the process implementing the single-level store has a number of user threads that we cannot influence. These threads are not supposed to use any system calls that result in explicit i/o to the single-level store, such as `read` and `write`. They might use such system calls on sockets that are not connected to disk files.

At any point in time, parts of the file are mapped into the address space (using mmap). The vast majority of virtual pages are mapped to the corresponding locations in the data area in the file. These pages are write protected. Some pages are mapped read/write into the primary zone. Yet some more are mapped write protected to pages in the secondary zone.

We have no control over page faults, because our process might be competing with others. We detect only write faults. Write faults do not occur on pages in the primary zone, because pages mapped to this area are not write protected. Thus, a write fault occurs as a result of trying to modify a page that is either in the secondary zone, or in the data area.

When a write fault occurs, we allocate space for the new version of the page (by moving the first page in the free zone to the primary zone). If the page is potentially dirty (which we notice by looking it up in the secondary address map, though we cannot know for sure that it is dirty as the system might have written it to the secondary zone) then we use `msync` to make sure it is written to that area. It is possible then, that this call to msync does nothing. We must now change the address map (by using mmap) of the process so that this in-core page is mapped to the newly allocated location in the primary zone.

To do that, we copy the contents of the page to a secret location (a single global page which is not part of the normal address space of the process), then unmap

(using `munmap`) the page, map it (using `mmap` read/write) to the new location in the primary zone, and finally copy the page contents from the secret place back to the page.

If the page is not potentially dirty, either because it is in the secondary address map but has not been written to after it was saved, or if it is in the data area, we copy it to secret location, unmap it, map it to the new location and copy it back.

Write faults are handled by the mutator threads. This might require some locking to avoid race conditions.

In addition to mutator threads, there is a thread responsible for checkpointing. A global variable indicates whether the system is in the save state or in the migrate state.

When the system is in the save state, the checkpointing thread scans the in-core secondary address map for potentially dirty pages (we do not know whether they are really dirty) and calls msync on such pages. Synchronization is required since we might be in competition with mutator threads. When no more potentially dirty pages exist, we write the secondary address map to disk. We could either use lseek and write, or mmap the address maps into the address space. Finally the initial page is written (or msynced). Following the update on disk, the tertiary zone is incorporated into the free zone, and the state is changed to migration.

In the migrate state, the checkpointing process scans the secondary address map. For each page it finds, it checks whether the page also occurs in the primary map. If that is the case, the entry is simply deleted from the secondary address map. In this case, the memory map of the process has the page mapped to a location in the primary zone. If the page does not occur in the primary address map, it should be migrated to its home location (as indicated by the map). At this point, the home location is not mapped into the address space of the process, so we use lseek+write to write the contents to the home location in the file. When the secondary address map is empty, the system is ready for a new checkpoint.

To declare a new checkpoint, all pages that have entries in the primary address map must be atomically write protected. In order to do that, we must first stop all the mutator threads, then call mprotect on each of the pages in the primary address map, reassign the current secondary zone to be tertiary, the primary address map becomes secondary, and a new primary address map is created, the system is put into the save state and the mutator threads are restarted.