CLOSOS
Specification of a Lisp operating system.

Robert Strandh

2013

Contents

Chapter 1

Introduction

1.1 What a Lisp operating system is

A Lisp Operating System (LispOS for short) is not just another operating system that happens to be written in Lisp (although that would be a good thing in itself). For the purpose of this document, a LispOS is also an operating system that uses the Lisp interactive environment as an inspiration for the interface between the user and the system, and between applications and the system.

In this document, we give some ideas on what a LispOS might contain, how it would be different from existing operating systems, and how such a system might be created.

1.2 Problems with existing systems

1.2.1 The concept of a *process*

Most popular existing operating systems are derived from UNIX which was written in the 1970s. The computers for which UNIX was intended had a very small address space; too small for most usable end-user applications. To solve this problem, the creators of UNIX used the concept of a *process*. A large application was written so that it consisted of several smaller programs, each of which ran in its own address space. These smaller programs would communicate by having one application write text to its output stream for another application to read. This method of communication was called a *pipe* and a sequence of small applications was called a *pipeline*. As a typical example of a chain of applications, consider the pipeline for producing a typeset document (one of the main applications for which UNIX was designed). This chain had a program for creating tables (called `tbl`), a program for generating pictures (called `pic`), a program for generating equations (called `eqn`), and of course the typesetting program itself (called `troff`).

The computers that UNIX was intended to run on did not have any memory-management unit (MMU). The absence of memory management meant that the code could not move around in physical memory depending on whether other programs were present in memory as well. To solve this problem, a mechanism called *swapping* was used. Each program was written so that it had the entire physical address space at its disposal, and to make that work, one process at a time was present in physical memory. To give the illusion of multi-programming, at regular intervals the current process was interrupted, moved from main memory to secondary memory, and another runnable process was loaded into main memory instead. Programs written in low-level languages such as C and C++ are still written as if they were meant to be executed on such early computers.

Using UNIX-style pipes to communicate between different components of an application has several disadvantages:

- To communicate complex data structures (such as trees or graphs), they must be converted to a stream of bytes by the creating component, and it must be analyzed and parsed into an equivalent data structure by the using component. Not only is this unparsing/parsing inefficient in terms of computing resources, but it is also problematic from a software-engineering point of view, because the external format must be specified and maintained as a separate aspect of each component.

- An artificial *order* between the different components is imposed, so that components can not work as libraries that other components can use in any order. Sometimes (as in the example of the `troff` chain) the end result of a computation depends in subtle ways on the order between the components of the chain. Introducing a new component may require other components to be modified.

Pipes also have some advantages though. In particular, they provide a *synchronization* mechanism between programs, making it very easy to implement producer/consumer control structures.

It is an interesting observation that in most text books on operating systems, the concept of a process is presented as playing a central role in operating-system design, whereas it ought to be presented as an unfortunate necessity due to the limited address space of existing computers in the 1970s. It is also presented as *the* method for obtaining some kind of *security*, preventing one application from intentionally or accidentally modifying the data of some other application. In reality, there are several ways of obtaining such security, and separate address spaces should be considered to be a method with too many disadvantages.

Nowadays, computers have addresses[1] that are 64 bit wide, making it possible to address almost 20 exabytes of data. To get an idea of the order of magnitude of such a number, consider that a fairly large disc that can hold a terabyte of data. Then each byte of 20 million such discs can be directly addressed by the processor. We can thus consider the problem of too small an address space to be solved. The design of CLOSOS takes advantage of this large address space to find better solutions to the problems that processes were intended to solve.

1.2.2 Hierarchical file systems

Existing operating system come with a *hierarchical file system*. There are two significant problems, namely *hierarchical* and *file*.

The *hierarchy* is also a concept that dates back to the 1970s, and it was

[1]The virtual address is 64 bits wide. That does not mean that all 64 bits are used on all implementations of the architectures. However, on the current (as of this writing) Intel and AMD x86-64 processors, at least 48 bits are used, and this number is likely to increase in the future.

considered a vast improvement on flat file systems. However, as some authors[2] explain, most things are not naturally hierarchical. A hierarchical organization imposes an artificial order between names. Whether a document is called `Lisp/Programs/2013/stuff`, `Programs/Lisp/2013/stuff`, or something else like `2013/Programs/Lisp/stuff`, is usually not important.

The problem with a *file* is that it is only a sequence of bytes with no structure. This lack of structure fits the UNIX pipe model very well, because intermediate steps between individual software components can be saved to a file without changing the result. But it also means that in order for complex data structures to be stored in the file system, they have to be transformed into a sequence of bytes. And whenever such a structure needs to be modified by some application, it must again be parsed and transformed into an in-memory structure.

1.2.3 Distinction between primary and secondary memory

Current systems (at least for desktop computers) make a very clear distinction between primary and secondary memory. Not only are the two not the same, but they also have totally different semantics:

- Primary memory is *volatile*. When power is turned off, whatever was in primary memory is lost.
- Secondary memory is *permanent*. Stored data will not disappear when power is turned off.

This distinction coupled with the semantics of the two memories creates a permanent conundrum for the user of most applications, in that if current application data is *not* saved, then it will be lost in case of power loss, and if it *is* saved, then previously saved data is forever lost.

Techniques were developed as early in the 1960s for presenting primary and secondary memory as a single abstraction to the user. For example, the Multics system had a single hierarchy of fixed-size byte arrays (called segments) that served as permanent storage, but that could also be treated as any in-

[2] See `http://www.shirky.com/writings/ontology_overrated.html`

memory array by applications. As operating systems derived from UNIX became widespread, these techniques were largely forgotten.

1.2.4 Full address-space access

With operating systems such as UNIX, programs written in low-level languages such as C are written so that they have access to the full (virtual) address space[3] except that such a program naturally can not access the contents of a virtual address that does not have any physical memory associated with it.

Programs are written like that for historical reasons. Early computers had no memory-management unit, so there was no way to prevent a program from accessing the contents of any address. Essentially, we still write programs today as if we were using computers with no memory-management unit.

Full address-space access is a notorious source of security problems, in particular in combination with a programming language like C. The C language specification leaves many situations unspecified, and most compilers take advantage of this freedom to optimize for speed, to the detriment of other aspects such as security. As a result, it is possible for C programs to construct arbitrary data and arbitrary addresses and alter large parts of its addressable memory in uncontrolled ways.

Thus if a program does not take great care to prevent a temporary buffer from overflowing, reading an external document such as a web page may overwrite part of the stack[4] (which is located in the address space of the process). Such a buffer overflow can alter the return address of the currently executing function, so that instead of returning normally, it returns to some code that can have an effect that the program was absolutely not meant to have. It can do that because the C library is linked into the same address space as the rest of the code, so anything that a program can do with the C library, such as deleting files or transfer sensitive information to an external computer, can be done as a result of reading an external document.

There have been attempts to mitigate these basic problems with a fully ac-

[3]Or sometimes half of it, the operating system kernel occupying the other half.

[4]Problems with buffer overflow are not limited to the stack, of course. Overflowing a buffer located on the heap is a security problem as well.

cessible address space. Recently, for instance, a technique called *address space layout randomization*[5] has started being used to prevent the problems caused by full address-space access. The technique consists of giving the code of the main program and of the libraries that it uses different virtual addresses each time the programs is executed. That way, a malicious document can not rely on the address to return to being at a particular location, and defective programs that do not check for buffer overflow can continue to exist without so much danger in terms of security.

But address space layout randomization has its own problems. For one thing, a program can no longer be written to have predefined data structures with absolute virtual address at start-up. Either relative addressing must be used (which complicates the code and thus makes it less maintainable), or such data structures must use symbolic addresses to be resolved by the dynamic linker at program start-up (which also complicates the code, but in addition slows down program start-up because of additional work that the linker must do).

In summary, then, a system in which a user program executes in a process with an address space to which the code has full access will always have problems in terms of security, performance, maintainability, or a combination of those.

1.2.5 The concept of a kernel

The kernel of an operating system is a fairly large, monolithic program that is started when the computer is powered on. The kernel is not an ordinary program of the computer. It executes in a privileged state so that it has full access to devices and to data structures that must be protected from direct use by user-level programs.

The very existence of a kernel is problematic because the computer needs to be restarted whenever the kernel is updated, and then all existing state is lost, including open files and data structures that reside in volatile memory. Some programs, such as web browsers, compensate somewhat for this problem by remembering the open windows and the addresses that were associated with each window.

The fact that the kernel is monolithic poses a problem; because, when code

[5] https://en.wikipedia.org/wiki/Address_space_layout_randomization

needs to be added to the kernel in the form of a kernel module, such code has full access to the entire computer system. This universal access represents a security risk, of course, but more commonly, the module can be defective and then it will fail often by crashing the entire computer.

The problem with traditional kernels compared to the planned LispOS described in this document is similar to the difference between an executable file resulting from a program written in C and a Common Lisp system.[6] In a traditional executable program created by the linker from a collection of modules, it is hard to replace an individual function. The linker has turned the entire program into a monolithic executable in which addresses have been resolved once and for all. Compare that situation to a typical Common Lisp system in which it is normal practice to replace a single function, redefine a class, or add a method to a generic function, without restarting the Common Lisp system. The planned LispOS will be able to have parts of it updated, just as an ordinary Common Lisp system is able to do, without rebooting.

We have had solutions to this problem for many decades. The Multics system, for example, did not have a kernel at all. An interrupt or a system call was executed by the user-level process that issued the system call or that happened to be executing when the interrupt arrived. The code that executed then was not part of a monolithic kernel, but existed as independent programs that could be added or replaced without restarting the system. The system could still crash, of course, if some essential system-wide data structure was corrupted, but most of the time, only the user-level process that issued the request would crash, simply because the problem was limited to the address space of a single process. Multics did not have a kernel, but it still had the problem of full access to its own address space, so that the stack could be overwritten by a defective end-user program.

1.2.6 Mediocre input/output performance

Recent research [BMPR17] [WH18] indicates that the performance of input and output in traditional kernel-based systems is not good enough for some of the modern devices now becoming available. Recall that, in order to perform

[6]Thanks to Daniel Kochmański for suggesting this comparison, and for letting me use it here.

some input or output, an application program must make a system call so that the kernel can perform the operation on behalf of the application. Things are organized this way in order to prevent application programs from directly accessing devices so as to protect those devices from getting incorrect controls. Thus, input and output requires a *context switch* which consists of the *system call* itself, a change of the *page table* for address translation, and *flushing the cache* since virtual addresses are no longer valid. Such a context switch typically takes around $1\mu s$.

For typical devices such as disks, performance is not a problem because these devices are very slow compared to the time it takes for the context switch. However, for some modern storage devices the slow context switch is a problem.

1.3 Objectives for a Lisp operating system

The three main objectives of a Lisp operating system correspond to solutions to the two main problems with existing systems as indicated in the previous section.

1.3.1 Single address space

Instead of each application having its own address space, we propose that all applications share a single large address space. This way, applications can share data simply by passing pointers around, because a pointer is globally valid, unlike pointers in current operating systems.

Clearly, if there is a single address space shared by all applications, there needs to be a different mechanism to ensure *protection* between them so that one application can not intentionally or accidentally destroy the data of another application. Many high-level programming languages (in particular Lisp, but others as well) propose a solution to this problem by simply not allowing users to execute arbitrary machine code. Instead, they allow only code that has been produced from the high-level notation of the language and which excludes arbitrary pointer arithmetic so that the application can only address its own data. We shall call this kind of system a *controlled access system*[7] and we shall call the typical modern operating system where a process has full access to its address space, an *arbitrary access system*.

In order for access to be completely controlled, some optimizations that current Common Lisp compilers allow, must be ruled out. Examples of such optimizations are avoiding array-bounds checking (typically when the `safety` quality is set to 0) or trusting the programmer with `dynamic-extent` declarations. Such optimizations could still be allowed in system code, but installing such code would require additional privileges, equivalent to those of system administrators on current operating systems.

It might sometimes be desirable to write an application in a low-level language like C or even assembler, or it might be necessary to run applications that have been written for other systems. Such applications could co-exist with the normal ones, but they would have to work in their own address space as with current operating systems, and with the same difficulties of communicating with other applications.

1.3.2 Object store based on attributes

Instead of a hierarchical file system, we propose an *object store* which can contain any objects. If a file (i.e. a sequence of bytes) is desired, it would be stored as an array of bytes.

Instead of organizing the objects into a hierarchy, objects in the store can optionally be associated with an arbitrary number of *attributes*. These attributes are *key/value* pairs, such as for example the date of creation of the archive entry, the creator (a user) of the archive entry, and the *access permissions* for the entry. Notice that attributes are not properties of the objects themselves, but only of the archive entry that allows an object to be accessed. Some at-

[7] In the literature, this technique is sometimes called "trusted compiler", be we want to avoid that terminology in this document, because it suggests that the compiler must somehow be formally verified correct in order for this technique to be useful. Technically, the typical modern operating system would then have to be formally verified correct in order for the separation of address spaces to be a trusted mechanism. Clearly, we use such modern operating systems on a daily basis without any such formal verification, and we are reasonably sure that it respects that separation.

tributes might be derived from the contents of the object being stored such as the *sender* or the *date* of an email message. It should be possible to accomplish most searches of the store without accessing the objects themselves, but only the attributes. Occasionally, contents must be accessed such as when a raw search of the contents of a text is wanted.

For a more detailed description of the object store, see Chapter 2.

It is sometimes desirable to group related objects together as with *directories* of current operating systems. Should a user want such a group, it would simply be another object (say instances of the class `directory`) in the store. Users who can not adapt to a non-hierarchical organization can even store such directories as one of the objects inside another directory.

When (a pointer to) an object is returned to a user as a result of a search of the object store, it is actually similar to what is called a "capability" in the operating-system literature. Such a capability is essentially only a pointer with a few bits indicating what *access rights* the user has to the objects. Each creator may interpret the contents of those bits as he or she likes, but typically they would be used to restrict access, so that for instance executing a *reader* method is allowed, but executing a *writer* method is not.

1.3.3 Single memory abstraction

Current computers have two kinds of memory, a *primary* memory which is fast, volatile, and expensive, and *secondary* memory which is slow, permanent, and cheap. In contrast, the Lisp operating system would present a single abstraction of the memory, which looks like any interactive Lisp system, except that data is permanent.

In an implementation of a Lisp operating system on a current computer with two kinds of memory, the primary memory simply acts as a *cache* for the secondary memory, so that the address of an object uniquely determines where in the secondary memory it is stored. The cache is managed as an ordinary *virtual memory* with existing algorithms.

There are some indications that future computers may feature new memory technology with is fast, permanent, and cheap. An implementation of a Lisp operating system on such a computer will have the same abstraction of the memory, but its structure will be greatly simplified.

Since data is permanent, application writers are encouraged to provide a sophisticated *undo* facility.

1.3.4 Other features

Crash proof (maybe)

There is extensive work on crash-proof systems, be it operating systems or database systems. In our opinion, this work is confusing in that the objective is not clearly stated.

Sometimes the objective is stated as the desire that no data be lost when power is lost. But the solution to that problem already exists in every laptop computer; it simply provides a *battery* that allows the system to continue to work, or to be *shut down* in a controlled way.

Other times, the objective is stated as a protection against defective software, so that data is stored at regular intervals (checkpointing), perhaps combined with a *transaction log* so that the state of the system immediately before a crash can always be recovered. But it is very hard to protect oneself against defective software. There can be defects in the checkpointing code or in the code for logging transactions, and there can be defects in the underlying file system. We believe that it is a better use of developer time to find and eliminate defects than to aim for a recovery as a result of existing defects.

Multiple simultaneous environments

To allow for a user to add methods to standard generic functions (such as `print-object`) without interfering with other users, we suggest that each user gets a different *global environment*. The environment maps *names* to *objects* such as functions, classes, types, packages, and more. Immutable objects (such as the `common-lisp` package)[8] can exist in several different environments si-

[8]The `common-lisp` package is probably a bad example of an immutable object, because it could very well be necessary to make modifications to it on a per-user basis as a result of

multaneously, but other objects (such as the generic function `print-object`) would be different in different environments.

Multiple environments would also provide more safety for users in that if a user inadvertently removes some system feature, then it can be recovered from a default environment, and in the worst case a fresh default environment could be installed for a user who inadvertently destroyed large parts of his or her environment.

Finally, multiple environments would simplify experimentation with new features without running the risk of destroying the entire system. Different versions of a single package could exist in different environments.

For more details on multiple environments, see Chapter 4.

Safe concurrency

Any modern operating system must be written to handle *concurrency*, both in terms of *context switches* at arbitrary times, but especially in terms of *multiple simultaneous threads* of execution resulting from the execution of the system on a computer with multiple cores.

In particular, we will guarantee the integrity of the system in the presence of concurrency, so that there are no race conditions that may cause the system to be in an undefined state. We accomplish this guarantee by well known techniques such as locks, lock-free data structures, transactional memory, etc.

Furthermore, the global system garbage collector (See Section 5.), will itself be parallel and concurrent in order to take advantage of the existence of multiple cores, and in order to minimize pauses during garbage collection.

1.4 How to accomplish it

The most important aspect of a Lisp operating system is not that all the code be written in Lisp, but rather to present a Lisp-like interface between users and the

the installation of different software systems.

system and between applications and the system. It is therefore legitimate to take advantage of some existing system (probably Linux or some BSD version) in order to provide services such as device drivers, network communication, thread scheduling, etc.

1.4.1 Create a Lisp system to be used as basis

The first step is to create a Common Lisp system that can be used as a basis for the Lisp operating system. It should already allow for multiple environments, and it should be available on 64-bit platforms. Preferably, this system should use as little C code as possible and interact directly with the system calls of the underlying kernel.

1.4.2 Create a single-user system as a UNIX process

In parallel with creating a new Common Lisp system, it is possible to implement and test many of the features of the interface between the system and the users, such as the object store (probably without access control) using an existing Common Lisp system running as a process in an ordinary operating system.

The result of this activity would be sufficient to write or adapt several applications such as text editors, inspectors, debuggers, GUI interface libraries, etc. for the system.

1.4.3 Create a multi-user system as a UNIX process

With the new Common Lisp system complete and the object store implemented, it will be possible to create a full multi-user system (including protection) as a UNIX process, where the UNIX system would play the role of a virtual machine, supplying essential services such as input/output, networking, etc.

1.4.4 Create a bootable system

The final step is to replace the temporary UNIX kernel with native device drivers, and to write the code for required system services such as the *thread scheduler*, *synchronization primitives*, etc. Such a system could initially run in an emulator such as QEMU in order to facilitate debugging. Integration with an existing operating system could be accomplished by communication with the host operating system through its X11 server, which would avoid the necessity of a native display server for the Lisp operating system.

Chapter 2

Object store

The *object store* is a system-wide database containing any kind of objects. Each object is a *capability*.

An object in the store can optionally be associated with a certain number of *attributes*. An attribute is a *pair* consisting of the *attribute name* and the *attribute value*. The attribute name is a symbol in the `keyword` package. The attribute value can be any object.

Keyword	Possible values
category	The nature of the object such as **movie**, **music**, **article**, **book**, **user manual**, **dictionary**, **course**, **lecture**, **recipe**, **program**, **bank statement**, **email**. These would be chosen from an editable set that is defined per user.
name	A string that is displayed with the object, such as "A Dramatic Turn of Events", "Three seasons", "Alternative energy".
author	An object identifying a person, an organization, a company, etc.
genre	**progressive metal**, **science**, **algorithms**, **garbage collection**, **game**, **programming language implementation**, **operating system**. These would be chosen from an editable set that is defined per user.
format	This attribute can be used to identify the file type of documents such as **PDF**, **ogg/vorbis**, **MPEG4**, **PNG**, in which case the attribute can be assigned automatically, but also to identify the source format of files in a directory containing things like articles or user manuals, for example **LaTeX**, **Texinfo**, **HTML**. These would be chosen from an editable set that is defined per user.
date of creation	A date interval.
composer	An object representing a person. On a compilation album there can be more than one attribute of this kind.
language	An object representing a natural language such as **English**, **Vietnamese**, or a programming languages such as **Lisp**, **Python**. These would be chosen from an editable set that is defined per user. If appropriate, a document can have several of these attributes, for instance if some program uses multiple programming languages, or if a document is written using several languages, such as a dictionary.
duration	An object representing a duration.
source control	**GIT**, **SVN**, **CVS**, **darcs**, etc. These would be chosen from an editable set that is defined per user.

In a typical operating system installation, there are many fairly large objects such as movies, music files, pictures, etc. The amount of data associated with such an object that would be stored in the object store is typically very small compared to the object itself. Even a fairly modest text file probably has $10^4 - 10^5$ characters in it, whereas the meta-data probably takes up no more than $10^2 - 10^3$ bytes. It is therefore likely that the entire object store will fit in main memory. Scanning the entire object store would then take at most a few second of CPU time. For better performance, one or more *indexes* could be created. The objects could for instance be divided by *category*.

Searching the object store amounts to defining a *filter*, i.e. a function that, given a set of keyword/value pairs, returns *true* if and only if the corresponding object should be included in the search result. The result is returned to the user in the form of a *directory object* which is a list of *object entries* where each entry contains the object itself and the attributes of the object from the store, if any.

Chapter 3

Protection

There are two kinds of protection that are important in an operating system:

- *protecting different users from each other.* User A should not be able to access or destroy the data of some other user B, other than if B explicitly permits it, and then only in ways that are acceptable to B.
- *protecting the system from the users.* Users should be able to access system resources such as memory and peripherals only in controlled ways, so as to guarantee the integrity of the system.

3.1 Protecting users from each other

We use a combination of *access control lists* and *capabilities.* All heap-allocated objects except `cons` cells and (heap-allocated) numbers are manipulated through a *tagged pointer.* In addition to containing a type tag, the pointer also contains an *access tag.* The access tag consists of the 4 most-significant bits of a 64-bit pointer. Before a pointer is used to fetch an object from memory, the access bits are cleared. A primitive operation to fetch the access tag of a pointer is available to any user code. Each of the 4 bits represents a potential *access restriction*, the significance of which is up to the programmer. A function that wishes to restrict permission to some object can test the corresponding access bit and signal an error if that bit is set.

The author of some complex data structure may for instance grant access to it only to certain other users. This would be done by interpreting one of the access bits as *read permission*, and by having generic functions that access the data structures check that this bit has the desired value (for instance in a `:before` method).

The access bits of a capability are determined when the object is accessed through the object store. (See Chapter 2.) One of the possible attributes associated with the object in the object store corresponds to the access permissions in the form of an *access control list*. A user who accesses the object from the object store will be checked against the access control list and appropriate access bits will be cleared in the object before it is given to the user.

3.2 Protecting the system from the users

In a typical modern operating system, the system is protected from the users through the use of a *mode* of execution of the processor, which can be either *user mode* or *supervisor mode*. Certain instructions are restricted to supervisor mode, such as instructions for input/output or for remapping the address space.

In CLOSOS, the normal mode of execution is *supervisor mode*. The code executed by the user is translated to machine code by a compiler which is known not to generate code that, if executed, might represent a risk to the integrity of the system. Since no remapping of the address space is required as a result of an *interrupt* or a *trap*, such events can be handled very quickly.

Occasionally, it might be useful to write or install some software that is compiled to machine code by some compiler that does not necessarily generate code with controlled access, such as a compiler for some typical low-level programming language used today. The result of such a compilation or installation is a single (possibly large) Lisp function. When this function is executed, the mode of execution is switched to *user mode*. As with traditional modern operating systems, the code of such software has its own *address space*, which means that it can not directly manipulate CLOSOS capabilities. Instead, it has to communicate with the system through the user of *system calls*. A system-wide object is referred to by such code through an interposing *object descriptor*, much like a file descriptor in UNIX. The details of this mechanism have not yet been fully determined.

Chapter 4

Environments

Recall that an *environment* is a mapping from *names* to *objects*. This mapping consists of a set of *bindings*.

When a user is created in the system, a *default global environment* is created for that user. The global environment of a user consists of a *system-wide* environment and a *user-specific* environment.

The system-wide environment consists of bindings that are themselves immutable (i.e., the user is not allowed to alter the binding) such as the binding of the symbol `cl:length` to the function that returns the length of a sequence.[1] The objects of these bindings are also immutable, such as the length function itself. The system-wide environment is the same for every user, allowing the installation of software that is immediately visible to all users.

The user-specific environment consists of bindings that are created by the user. These bindings are of three different kinds:

- Bindings created by the user for instance as a result of executing a

[1]It may be necessary to allow the user to change bindings such as the one of `cl:length` to the function that returns the length of a sequence. In fact, it may be necessary to allow the user to modify every binding, in which case the global environment for a user contains no *system-wide* environment. Alternatively, the system-wide environment would be reduced to a small set of bindings. Perhaps bindings that allow the user to recover after destroying his or her environment should be stored there.

`defparameter` or `defun` form.

- Default system-wide bindings that can be altered by the user, such as the value of `*print-base*`.
- Immutable bindings where the *object* can be modified by the user, such as system-defined generic functions to which the user is allowed to add specific methods. Each user has a private copy of such objects.

The environment contains the following mappings:[2]

- Mappings from names to *packages* as managed by `make-package`, `delete-package`, `defpackage`, etc.
- Mappings from names to *function objects*, as managed by `symbol-function`, `(setf symbol-function)`, `fdefinition`, and `(setf fdefinition)`.
- Mappings from names to *macro functions*, as managed by `macro-function` and `(setf macro-function)`.
- Mappings from names to compiler macros, as managed by `compiler-macro-function` and `(setf compiler-macro-function)`.
- Mappings from names to *classes* as managed by `find-class`, `(setf find-class)`, `defclass`, `defstruct`, `define-condition`, etc.
- Mappings from names to *type definitions* established by `deftype`.
- Mappings from names to *global symbol macros* defined by `define-symbol-macro`.
- Mappings from names to *constant variables* defined by `defconstant`.
- Mappings from names to *special variables* defined by `(proclaim special)`, `defvar`, etc.

When a function or method object is created as a result of calling `compile` on a lambda expression, or as a result of loading a *fasl* file, the object is *linked* to the current global environment, in that external references are then resolved. When

[2] I may have forgotten some mappings that are part of the global environment.

such a function or method object is given to a different user, that different user can execute it, but external references in it will still refer to the environment into which it was compiled or loaded.

Notice that methods are not in themselves part of the environment. When we say that a method is *linked* to the current global environment, we just mean that references to symbols within that method are resolved in the current global environment.

This mechanism provides an efficient method of protection. User A can grant controlled access to part of his or her global environment by allowing a user B to execute a function made available to him or her through the *object store*. (See Chapter 2.) In a traditional modern operating system such as UNIX, this kind of controlled access required the use of the *setuid* mechanism, simply because in such a system there is no way to access an object other than through the global file system, and the accessing user must have the right permissions to access the object.

The same mechanism can be used by the system itself to protect objects that would be unwise to give users direct access to, such as disks or printers.

Chapter 5

Garbage collection

5.1 Introduction

Contrary to traditional operating systems such as UNIX, a Lisp operating system will need a global *tracing garbage collector*. Traditional operating systems get away with a simpler technique, because the file system in such an operating system can not contain cycles. With this restriction, the simpler `reference counting` mechanism is sufficient. Furthermore, although reference counting is usually slower than a tracing garbage collector, the additional overhead of reference counters is of no importance when used for a file system in secondary memory.

There is a rich literature on automatic memory management. (see e.g., [JHM11])

For CLOSOS, we plan to have a two-level memory management technique. The low level consists of a relatively small local heap for each thread, and a per-thread garbage collector that manages that heap. The higher-level consists of a global heap that contains long-lived objects and objects that are shared between several threads.

5.2 Per-thread garbage collector

Each thread has a local heap, roughly the size of the cache, say around 4MiB. The thread-local heap is managed entirely by the thread itself, so that the garbage collector for it is executed by the thread itself. Experiments show that we will be able to run the thread-local garbage collector in a few milliseconds, which is good enough for most applications. We will use a sliding garbage collector in order to maintain allocation order. This way, we have a precise measure of the relative age of the objects, so that we can promote only the oldest objects when required.

There can be no references between an object in one local heap to an object in another local heap. And there can be no references from the global heap to a local heap. Whenever a reference is about to be created from an object in the global heap to an object in the local heap, this attempt is caught by a *write barrier* on the global heap. As a result if this write barrier being tripped, the object in the local heap being referred to (and its transitive closure) will migrate to the global heap, thereby preserving the general invariant.

5.3 Global garbage collector

In addition to the thread-local heaps, there is a global heap. The garbage collector for this heap will use a combination of the traditional *mark-and-sweep* collector and an ordinary memory allocator, similar to the one used by the C functions `malloc` and `free`.

Recall that that a heap-allocated object is either a `cons` cell or a *general instance*. A `cons` cell is represented as two machine words. A general instance is represented as a *header* consisting of two machine words, and a *rack* which is a vector of words with a contents that depends on the exact type of the object. In both cases, then, a reference to a heap-allocated object is a reference to a double word.

Given this representation, we separate the headers from the racks, so that `cons` cells and headers of general instances are allocated from a separate part of the global heap. Since this part of the global heap consists of only two-word objects, it can be managed very efficiently with a *mark-and-sweep* garbage collector, using a simple free list. The advantage of this technique is that an object is never moved as a result of a garbage collection. Therefore, any reference to an object that is shared between several threads, remains valid after a garbage collection of the global heap.

The marking phase is done by first requesting each thread to do a garbage collection and to mark any object in the global heap that is referred to by local objects. When all threads have responded, a global collection is started. The global collection is done concurrently with thread activity. For that reason, objects allocated in the global heap during this phase are marked as being live. The global collection traces the global heap starting with objects marked by the mutator threads. This tracing uses a standard three-color algorithm. Write operations to the global heap are caught by a write barrier.

When tracing in the global heap is finished, the part of the global heap that contains two-word headers and `cons` cells is scanned and unmarked cells are collected into a free list. If an unmarked cell is a `cons` cell, then no further action is needed. If an unmarked cell is a header object, then the corresponding rack is returned to the rack part of the global heap.

Chapter 6

Checkpointing

In this chapter, we describe two alternative checkpointing techniques. The first one is inspired by the work on the EROS operating system. The second one is based on work on log-structured file systems.

6.1 Technique inspired by EROS

The checkpointing mechanism described in this section is inspired by that of the EROS system.

The address of an object can be considered as consisting of two parts: the *page number* and the *offset within the page*. The page number directly corresponds to the location on disk of the page. However, when checkpointing is activated, the available disk memory is divided into three parts, and the page number should be multiplied by 3 to get the first of three disk locations where the object might be located.[1]

Checkpointing is divided into *cycles* delimited by *snapshots*. At any point in time, two checkpointing cycles are important. The *current* checkpointing cycle started at the last snapshot and is still going on. The *previous* checkpointing

[1]The price to pay for checkpointing is thus that disk memory will cost a factor 3 as much compared to the price when no checkpointing is used.

cycle is the one that ended at the last snapshot.

A page can exist in one, two, or three *versions*, located in three different places on disk. Version 0 of the page is the oldest version, and also the version that would be used when the system is rebooted after a crash. Version 0 of the page always exists. Version 1 of the page corresponds to the contents of the page as it was at the end of the *previous* checkpoint cycle. Version 1 of the page exists if and only if the page was modified during the previous checkpoint cycle. Version 2 of the page is the *current* version of the page. Version 2 of the page exists if and only if the page has been modified since the beginning of the *current* checkpoint cycle. We use the word *page instance* to refer to a particular version of a particular page.

A page can be associated with a *frame*.[2] An attempt to access a page that is not associated with a frame results in a *page fault*. At most one version of a particular page can be associated with a frame, and then it is the version with the highest number. A frame associated with version 0 or version 1 of a page is *write protected*, but a frame associated with version 2 of a page is not. Any attempt to modify the contents of a write-protected frame results in a *write fault*.

A frame can be *clean* or *dirty*. By definition, when the frame is clean, its contents are identical to those of the associated page instance. When the frame is dirty, it means that it has been modified after it was associated with the underlying page instance. A frame that is associated with version 0 of a page can not be dirty. If a frame that is associated with version 1 of a page is dirty, then it is because it was modified during the *previous* checkpointing cycle, and not the current one.

When a page fault occurs, and there are unused frames, an arbitrary unused frame is associated with the latest version of the page. If there are no unused frames when a page fault occurs (which is the normal situation), a frame that is already associated with a page must be freed up. To select the frame to free up, an ordinary ALRU method can be used. If the selected frame is dirty, the contents are written to the page instance associated with the frame. Finally, the latest version of the requested page is associated with the selected frame. If the latest version of the requested page is either version 0 or version 1, then

[2] A *frame* is the main-memory instance of a page.

the frame is write protected before execution resumes.

As indicated above, when a write fault occurs, the frame written to must be associated with either version 0 or version 1 of a page. If it is associated with version 0 of the page, then the frame must be clean. In that case, the association of the frame is modified, so that it henceforth is associated with version 2 of the page. Before execution resumes, the frame is unprotected. As soon as execution resumes, the frame will be marked as dirty since the reason for the fault was an attempt to write to it. When a write fault occurs and the frame is associated with version 1 of the associated page, the frame may be either clean or dirty. If it is clean, again, the association of the frame is modified so that it henceforth is associated with version 2 of the page, and again the frame is unprotected before execution resumes. If the frame is dirty, then its contents are first written to the associated page instance. Then the association is changed as before.

To determine the disk location of each version of each page, we use a *version table*. The version table is just a sequence of bytes, one for each page. Only 6 bits in each byte are actually used. The two least significant bits indicate the location of version 0 of the page. 00 means the first of the 3 possible consecutive disk locations, 01 means the second and 10 means the third, and 11 is not used. The next two bits indicate the location of version 1 of the page, with the same meaning as before, except that 11 means that there is no version 1 of the page. The final two bits indicate the location of version 2 of the page with the same interpretation as for version 1.

At any point in time, there exist three version tables; two on disk and one in main memory. The two versions on disk play the same role as the disk tables in EROS, i.e., while one of them is being updated, the other is still complete and accurate. A single bit in the boot sector of the disk selects which one should be used at boot time. When a new version table needs to be written to disk, it is first written to the place of the unused disk table, and then the boot sector is written with a flipped selection bit.

The version table in main memory is represented in two levels with a *directory* of pages. If one page is 4kiB, then one page can hold 2^{12} version table entries. For a 300GB disk (with room for around 25 million pages), the directory will contain around 6000 entries. A directory entry contains not only a pointer to the page of table entries, but also a bit indicating whether any of the table entries in the corresponding page indicates a page which exists in more than one version. It is expected that a relatively small fraction of the directory entries in each checkpointing cycle with have the bit set.

When a write fault occurs and as a result a new version of a page is created, the in-memory version table is consulted. The entry for the page indicates the disk location of version 0 of the page, and sometimes also version 1 of the page. The disk location for the new version (version 2) of the page is chosen to be one of the two unused ones (if only version 0 of the page exists) or the only unused one (if both version 0 and version 1 of the page exists). The location for version 2 of the page is indicated in the version table entry by setting bits 4 and 5 of the entry to the corresponding disk location.

In parallel with mutator threads, one or more threads scan the page table of the operating system for dirty frames. When a dirty frame corresponding to version 1 of a page is found, the contents of the frame is saved to its associated page instance, and the dirty-bit is cleared. When there are no more dirty frames corresponding to version 1 pages, the set of page instances corresponding to all version 1 pages and version 0 pages where no version 1 exists represents the state of the system at the time of the last snapshot.

To save the coherent state of the system to disk, the in-memory version table directory is scanned. Whenever a directory entry with the bit indicating the existence of pages with several versions set, the page of the directory entry is saved to disk. When the entire version table has been scanned, a new boot sector is written to indicate that the newly saved table is the current one.

The final action to take in order to finish the current checkpointing cycle and begin a new one is an *atomic flip*. This atomic flip consists of turning all version 1 pages into version 0 pages and all version 2 pages into version 1 pages. To do that, mutator threads must be stopped. Then the in-memory version table is scanned. Whenever an entry is found that has a version other than 0 in it, it is modified. If both a version 1 and a version 2 exists, bits 2 and 3 of the entry are moved to position 0 and 1, bits 4 and 5 are moved to positions 2 and 3, and positions 4, and 5 are set to 11. If no version 1 exists, then bits 4 and 5 are moved to positions 2 and 3, and positions 4, and 5 are set to 11. Finally, mutator threads are restarted.

The easiest way to modify a version table entry is probably to create a 64-byte table in memory which, for each possible version of the existing version table entry gives the new version. Even though it would require a memory access, this table will quickly be in the cache, so access will be fast.

To get an idea of performance of the atomic flip, let us take a situation where the *working set* is no bigger than the size of main memory.[3] Furthermore, let us say that the size of main memory is $64GiB$ and that around half the pages of the working set are modified in a particular checkpointing cycle. If we assume that the modified pages are concentrated with respect to the version table directory, then we can ignore the time to scan the version table directory. To accomplish the flip, we then need to modify 2^{23} entries. If we assume modified entries are adjacent, we can load and store 8 of them at a time, requiring 2^{21} memory accesses. If a memory access takes around 10ns, the flip will take around 20ms.

The time for a flip can be made shorter by taking more frequent snapshots.

6.2 Technique based on log-structured file systems

To make the description more concrete, we imagine a secondary storage device consisting of around 2^{30} pages, each containing 2^{12} bytes. Recall that CLOSOS treats primary memory as a cache for secondary memory. Therefore, the pages on the secondary storage device can be considered as making up the complete address space of CLOSOS. As such, they have unique numbers, starting at 0. In the example system, the unique page number would occupy bits $41 - 12$ of a pointer.

However, with the technique described in this section, the unique page number does not correspond to any fixed location on the secondary storage device. Instead, the location of a particular page can vary over time. But when a page fault for a particular unique page number occurs, the location of the page on secondary storage must be known. For that reason, we keep a *page map* in main memory. In the example system, this page map would consist of 2^{30} 4-byte entries, for a total of 2^{32} bytes of main memory.

[3] If the working set is larger than the main memory, performance is likely to deteriorate for more fundamental reasons.

With the technique described in this section, the secondary storage device represents a very large *circular queue* where each element of the queue is called a *segment*. Such a segment represents a unit of checkpointing. New segments are added to the tail of the queue. Old segments are removed from the head of the queue as described below.

A segment consists of:

- a *header* containing metadata about the contents of the segment, and
- a certain number of pages that may have been modified since the previous checkpoint.

Again, to make the description more concrete, let us imagine that the number of pages in a segment is around 250 or so, for a total of around $1MB$ of page data. A segment is written as a unit to the secondary storage device. If that device is a disk, then the seek time and rotation delay of the disk will not significantly impact the transfer of the segment to the disk, because the size of the segment is sufficiently large that the data-transfer time will dominate.

Furthermore, it is advantageous to keep the secondary storage device nearly full, because then (if the device is a disk) the head and the tail of the queue will be physically close, thereby minimizing seek time.

The header of a segment contains:

- A list of the unique page number of each of the pages in the segment. For the example segment size, this information occupies around $1KB$.
- A SHA value calculated from the data in the segment.
- The position of the head of the queue, i.e. the position of the first segment to be removed from the secondary device.

In addition to the queue of segments, the secondary storage device contains a single word of information, indicating the tail of the queue, i.e. the position on the device of the last checkpoint segment that was written.

The first thing we need to verify at this point is that it is possible to boot the system, given only the information on the secondary storage device. Here is how the system would be booted:

1. Read the information indicating where the tail of the queue is located.
2. Using this information, read the metadata of the last checkpointing segment that was written.
3. From this metadata, retrieve the information about the head of the queue.
4. Read each segment from the head to the tail of the queue, constructing the page map from the metadata of each segment.
5. Load initial pages into main memory, setting up the page tables as appropriate.
6. Jump to the entry point of the system.

Segments are removed from the head of the queue, by a procedure called *cleaning*. This procedure will be described later. For now, we assume that it is not present.

The system maintains three buffers, each one the size of a segment. Two buffers are used to alternate, so that one is being written to secondary memory while the other one (the *active one*) is used to receive pages in main memory. The third buffer is used to read back and compare what was written to secondary storage. Two counters, M and N, each with an initial value of 0 is kept for each of two ordinary segment buffers. M indicates the first free page in the active segment buffer, or equivalently, the number of pages that have already been copied to the buffer. N indicates the number of dirty pages that have not yet been copied to the segment buffer. If ever $M + N$ reaches the value corresponding to the number of pages in the buffer (in our example, 250, then a *checkpoint* is triggered as described below.

When a page fault occurs, a victim page is chosen using some standard technique, such as “least recently used”. If the victim page is clean, it is simply discarded and the page map is modified to reflect the change. If the victim page is dirty, its contents is copied to the first free page of the active segment buffer, and the value of M is incremented. The unique number of the page is retrieved from the page map and stored in the header of the active segment buffer.

All clean pages are read-only. When an attempt is made to modify a page, N is incremented and the page is marked as writable.

As mentioned above, when $M + N$ reaches the value corresponding to the number of available pages in the segment buffer, a checkpoint is triggered. The initial operation of a checkpoint is called an *atomic flip* which involves two segment buffers that we shall call A and B. A is the current active segment buffer with $M_A + N_A$ having reached its ceiling and B is the next one to be activated with its M_B and N_B equal to 0.

First, the N_A dirty pages not yet in the buffer are marked as read-only. This operation must be done atomically, i.e., all executing threads must be temporarily stopped. The active segment buffer is then set to segment B.

Then the N_A pages that were dirty are copied to segment buffer A. Their respective unique page numbers are retrieved from the page map and copied to the header of segment buffer A. Once this is done, the entire segment A is written to the end of the queue on secondary storage, and M_A and N_A are set to 0.

To avoid that the secondary storage device fills up with more and more checkpoint segments, an activity called *cleaning* works in parallel with the activity described above. Conceptually, a segment is read from the head of the queue and processed as follows. The list of unique page numbers in the segment header is examined. For each unique page number, the page map in main memory is consulted. There are two possible outcomes:

1. The location of the page as indicated by the page map is different from the location in the segment being processed. Then, there is a segment further back in the queue that contains a newer version of the page. Therefore, this version of the page is obsolete, and is simply discarded.

2. The location of the page as indicated by the page map is the same the location in the segment being processed. Then, this version of the page is the most recent one. In this case, the page is copied to the active segment buffer and M is incremented.

When every page in the head segment has been processed this way, the header of the active segment buffer is updated to reflect that the complete segment at the head of the queue has been processed and the following segment on the queue should be processed next. Notice that there is no danger in processing pages this way multiple times. Thus, if a crash occurs in the middle, there is no harm done.

Now, let us turn our attention to performance. Clearly, if a disk the size of the secondary storage device in our example is to be completely read when the system boots, it will take a very long time indeed. We suggest handling this problem by separating the segment headers from the segment pages either to two separate parts of a single storage device or to a second device. Only the headers need to be read for a page map to be constructed in memory. The headers are less than one half of a percent the size of the space occupied by pages in our example, so booting the system is then much faster. Even better, if the segment headers are placed on a persistent solid-state device, they can be read much faster.

Chapter 7

Device drivers

7.1 Introduction

The purpose of a device driver is to act as an intermediate layer between an operating-specific API that is common for a group of similar devices and vendor-specific interfaces for individual types of devices.

An important part of writing device drivers for a Lisp operating system is therefore to specify the different groups of devices and the corresponding operating-specific API for each group.[1]

7.2 Tickets

Some I/O operations, when called, return an object of type *ticket*. A ticket is either *pending* or it has *expired*. A pending ticket corresponds to an I/O operation that is not yet complete.

⇒ `ticket` [*Protocol Class*]

The base class for all tickets.

[1] As everything else in this document, this chapter is open to discussion. More so here, because I have no prior experience in defining device-driver APIs. – RS

⇒ `standard-ticket` [*Class*]

Instantiable subclass of the class `ticket`

⇒ `expired-p` *ticket* [*Generic Function*]

Return true if and only if *ticket* has expired.

⇒ `wait-some &rest` *tickets* [*Function*]

Suspend the current process until one of the tickets has expired.

⇒ `wait-all &rest` *tickets* [*Function*]

Suspend the current process until all of the tickets have expired.

7.3 Disk drivers

⇒ `disk` [*Protocol Class*]

This is the root class of all disk device classes. A disk is a device that stores data in *blocks*. The size of a block varies between different types of disks. Blocks are numbered from 0 to $N - 1$ where N is the total number of blocks on this device. Because of the existence of bad blocks, N may be smaller than the nominal size of the device. Nevertheless, the driver and the disk controller conspire to present the disk as contiguous sequence of blocks.

⇒ `standard-disk` [*Class*]

This class is an instantiable subclass of the class `disk`.

⇒ `size` *disk* [*Generic Function*]

Return the number of blocks that *disk* may store, so excluding bad blocks.

⇒ `block-size` *disk* [*Generic Function*]

Return the size of a native block for *disk*. This size is the preferred size to use in transfers to and from this type of disk.

⇒ `write-block` *disk block address* [*Generic Function*]

Issue a *write* operation transferring the data in *block* to *disk*. The parameter *disk* is an instance of the class *disk*, and *block* is a a vector of type `(simple-array (unsigned-byte 8) (*))`. The size of *block* must be a power of 2, and the address must be aligned to the size of *block*. If the size of *block* is *smaller* than the native block size of *disk* then a request to read an entire native block will be issued and the result will be stored in a temporary location. Part of the native block in the temporary location will then be overwritten by the contents of *block*. Finally, the contents of the temporary location will be written to the device. If the size of *block* is *greater* than the native block size of *disk*, then several native blocks will be written from *block*.

A call to this generic function returns an instance of the class `ticket`. The functions `wait-some` and `wait-all` can be used to wait for the I/O operation to finish.

⇒ `read-block` *disk block address* [*Generic Function*]

Write a block of data to the disk. The parameter *disk* is an instance of the class *disk*, and *block* is a a vector of type `(simple-array (unsigned-byte 8) (*))`. The size of *block* must be a power of 2, and the address must be aligned to the size of *block*. If the size of *block* is *smaller* than the native block size of *disk* then a request to read an entire native block will be issued and the result will be stored in a temporary location. Then a part of that native block will be copied to *block*. If the size of *block* is *greater* than the native block size of *disk*, then several native blocks will be read into *block*.

A call to this generic function returns an instance of the class `ticket`. The functions `wait-some` and `wait-all` can be used to wait for the I/O operation to finish.

Part I

Appendices

Appendix A

Use cases

In this appendix, we consider particular "use cases" or "scenarios", i.e. common situations that the user will need to handle. The purpose of this exercise is twofold:

1. To give the readers of this specification an idea of how the system might be used.
2. To determine the requirements of the object store (See Chapter 2.) based on real situations that it must handle, or not.

A.1 Opening a document for reading

This scenario is defined by the fact that the user wants to start the execution of some application, while giving it a particular document, presumably of the type that this application can handle. Examples of such situation are:

- The user wants to read a text document in PDF format.[1]

[1]Notice that PDF is a way of storing a structured document in a file consisting of a sequence of bytes. This is not the kind of document we mean here. We rather mean some structured version of the document containing the same sections as the PDF specification requires, but that is not stored as a sequence of bytes, but rather as a graph of instances of classes that together represent such a document.

- The user wants to watch a particular movie.
- The user wants to listen to some piece of music.

In all these cases, there are several ways in which the scenario can play out:

- The user might be interacting with a listener, and the user knows some Common Lisp form (perhaps the name of a special variable) to type in order to obtain the desired document. In this case, the user calls the top-level function of the application, passing it the result of the form as an argument.
- The user might be interacting with a listener, but the document is somewhere in the object store. Then the user first issues a request to the object store, perhaps with the document type and the title (or part of the title). The object store then presents[2] the documents that correspond to the query. Finally the user types the name of the application, but instead of giving a form as an argument, he or she clicks on the relevant presentation.
- The user might already be interacting with the right application. Therefore, he or she asks the application to read a different (or another) document. By doing this, the application starts a "document selector" that allows the user to either type a Common Lisp form with the new document as its value, or to issue a request to the object store. When the user selects the document, it becomes the reusult of the call to the document selector, and the application starts the execution on the new document.

[2]By "present", we mean that the output is in the form of CLIM presentations that are clickable.

Bibliography

[BMPR17] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011.

[WH18] Daniel Waddington and Jim Harris. Software challenges for the changing storage landscape. *Commun. ACM*, 61(11):136–145, October 2018.

Index