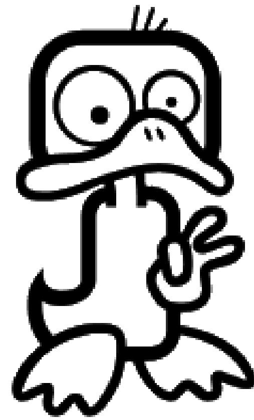
The kaneton microkernel



kaneton people

February 17, 2011

This document describes the kaneton microkernel research project design and implementation.

This document should be used by every student willing implement the kaneton educational microkernel as well as by people looking for more details on the kaneton microkernel design and implementation.

All the kaneton documents are available on the official website [1].

[1] http://kaneton.opaak.org

Contents

1 Introduction **5**

2 Background **7**
2.1 Distributed Operating System . . . 8
2.2 Microkernel . . . 9

3 Goals **11**

4 Design **15**
4.1 Managers . . . 16
4.2 Layers . . . 18

5 Implementation **21**
5.1 Interface . . . 22
5.2 Organisation . . . 23

6 Portability **25**
6.1 Background . . . 26
6.2 kaneton . . . 26
6.3 Interface . . . 27
6.4 Machine . . . 27

7 Boot **29**
7.1 Core . . . 31
7.2 Machine . . . 33

8 Licenses **35**
8.1 Pedagogical License . . . 36
8.2 kaneton License . . . 38

Chapter 1

Introduction

This chapter introduces the present document by drawing a list of the subjects discussed in this book.

The kaneton microkernel research project design and implementation are described through a number of documents. The first one in the chain is the present book called *The kaneton microkernel*. This book introduces the kaneton microkernel project through its goals, design, terminology and so forth.

The books listed below go further by describing more precisely a specific component of the kaneton microkernel:

1. **The kaneton microkernel :: design**

   This paper overviews the kaneton microkernel project by describing its concepts and principles;

2. **The kaneton microkernel :: core**

   That document focuses on the design and implementation of the kaneton microkernel core through the different managers;

3. **The kaneton microkernel ::** *[architecture]*

   Theses documents tackle the implementation of kaneton on the given *[architecture]*. Additionally, these documents contain information on platforms supporting this architecture.

   These documents can be found in two forms, either public or private. Indeed, these documents contain information students must not have access to. Therefore, the private version of these documents is not available and contains more details on the implementation of the given machine.

   The architectures, with their platforms, currently documented are listed below:

   - **ia32**: *ibm-pc*

Chapter 2

Background

This chapter introduces some notions about operating systems and kernels. However, this chapter assumes the reader already has a substential knowledge about operating systems internals.

2.1 Distributed Operating System

To understand the kaneton microkernel design and implementation, the reader must first understand what are the distributed operating systems requirements and more generally what are distributed systems inherent problems.

A distributed operating system can be defined through different ways and the reader should probably find different definitions in the literature. In this document, we assume that the main goal of a distributed operating system is to connect users and resources in a transparent, open, and scalable way.

For instance, when a user launches a program, the distributed operating system decides on which machine of the network to run it. More generally, for any action to perform, the distributed operating system tries to find the most suitable place for executing it depending on resources availability.

While the processes generally communicate with each other on the same machine, in a distributed operating system, they also communicate with the other processes of other machines as illustrated in *Figure 2.1*.

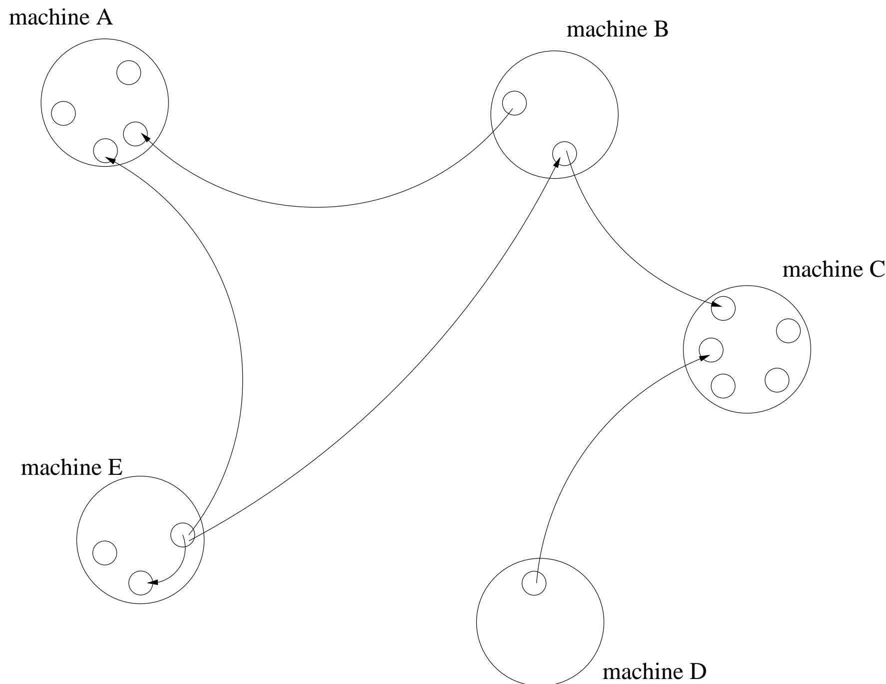


Figure 2.1: Communications in a distributed operating system.

In *Figure 2.1*, the machine named *E* is running three processes. One is just running some computations performing no communication, another is receiving a message from the latter which is also sending two messages to processes over the network.

These communications can be simple point-to-point communication for example between a client and a web server while other can be distributed operating system specific messages intended to make decisions on the location of a file on the distributed file system for instance.

We saw that machines on a distributed operating system are running processes which communicate with any other processes of any other node. Every machine needs an operating system to

organize the computer resources and to make them communicate.

Microkernel-based operating systems are the best suitable systems for building a distributed operating system as these systems are modular and reliable by nature.

2.2 Microkernel

A microkernel is a kernel type designed to be modular hence more reliable than monolithic kernels. Instead of building a large binary object containing the kernel itself, the drivers, the file systems *etc.* microkernels try to be as small as possible, providing only the fundamental services for managing memory, execution contexts, communication and input/output. Besides, the other classical operating system services are provided by userland processes with special privileges.

This very specific design is very interesting for security, maintainability and extensibility. Moreover, this type of design perfectly fits distributed operating systems requirements where special processes communicate with other node's processes to organize the whole distributed system resources.

Figure 2.2 illustrates a microkernel view with different layers.

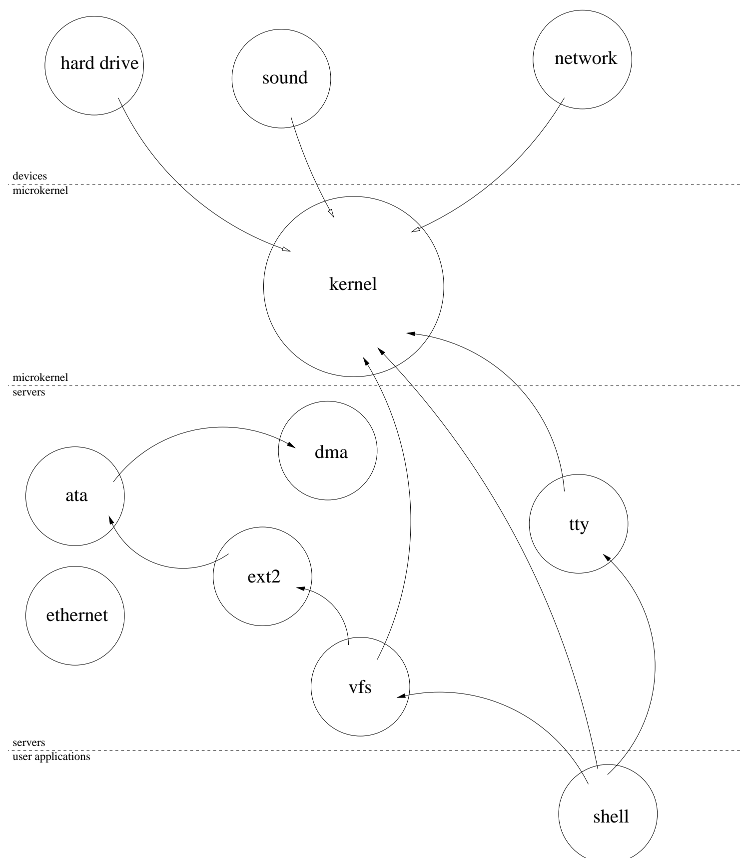


Figure 2.2: Communications in a microkernel hierarchy.

The whole microkernel design is based on the client-server communication model where clients ask servers to perform a specific task. For instance, to print some text to the screen, a user

program has to ask the *tty* server to do it for him because the user program itself does not have any right to perform such an action.

To avoid complications and deadlocks the microkernel follows a fundamental rule which restrict communication from clients only to more or equal privileged tasks. On the *Figure 2.2*, the *vfs* server can ask the *ata* server and the *kernel* to peform a task but the kernel cannot ask a less privileged server to do something for him.

Chapter 3

Goals

In this chapter we will briefly introduce the kaneton microkernel through the kaneton microkernel goals.

The project was primarily designed by two students in computer science, *Julien Quintard* and *Jean-Pascal Billaud*.

These two students previously actively contributed to the development of a nanokernel-based operating system project in a French research laboratory. This system was not powerful enough, especially from the design point of view.

Therefore, the two students started the design of a new microkernel by their own, called **kaneton**, for educational purposes.

The design was based on five fundamental guidelines.

1. **Educational**

   The kaneton project is built to become an educational project. The design as well as the implementation must therefore be as understandable as possible so that everyone interested in kernel internals can go through the documents and source code and actually understand how it works.

   This *understandable* property can be achieved through a very clear and coherent design. Moreover, the implementation should be written using modern tools and techniques to make the code as generic as possible and easily readable.

2. **Portability**

   The microkernel was particularly designed to be portable. The designers tried to develop a portability system powerful enough to port kaneton on any, existing or not, architectures.

3. **Maintanability**

   Although microkernel-based operating systems rely on a modular design, kaneton designers also wanted the microkernel itself to be modular and maintainable.

4. **Distributed Computing**

   The kaneton microkernel must be designed to fit distributed operating systems requirements. Indeed, the kaneton microkernel was developed in order to design and implement a distributed operating system named **kayou**.

   This point led to many specific choices in the kaneton microkernel design.

5. **Demystification**

   kaneton people wanted to break some well-known kind of computer science rules. Indeed, for instance, many computer scientists consider the source code as the actual documentation. Also, for many low-level programmers, the kernel boot source code and more generally the kernel source code itself cannot be understandable, clear and coherent as it is related to low-level programming: microprocessor, devices *etc.*

   kaneton people paid particular attention to the microkernel source code to be easily understandable, maintainable and extendable. Moreover, kaneton people tried to write documentation for every part of the project.

Notice that building an educational microkernel project is nothing innovative. Indeed few other projects already exist; the most popular being *MINIX* from *Vrije Universiteit*, *NachOS* from *Berkeley University* or *PintOS* from *Stanford University*.

kaneton people tried to design and implement a modern microkernel since, the original *MINIX* microkernel for example, do not use modern development tools. Moreover, the kaneton source

code is heavily commented and use modern languages techniques while trying to stay easily understandable.

The educational characteristic of kaneton does not constraint it from being optimised afterwards. kaneton people believe that implementing optimised algorithms in the first place does not lead to maintainable implementations.

Finally, note that the kaneton project is actually composed of two projects: the *kaneton microkernel* ***educational*** *project* which provides everything necessary to students willing to learn about kernels internals; and the *kaneton microkernel* ***research*** *project* which focuses on designing and implementing a powerful, reliable, flexible microkernel. Obviously these two projects are highly related as the kaneton educational project relies on the implementation of the kaneton research project.

Chapter 4

Design

This chapter introduces the kaneton design and its very specific terminology.

4.1 Managers

Although microkernel-based operating systems are, by nature, modular, kaneton people wanted the microkernel itself to be modular, subdivided into logical parts. This subdivision was introduced to make the whole microkernel clearer and more understandable.

The kaneton microkernel is thus divided into **managers**. These managers are generally responsible for a kaneton **object** type but there exist managers which manage something else or just create an abstraction over other kaneton managers. A kaneton object represents a logical and fundamental kernel entity. These objects are described later in this section.

Figure 4.1 illustrates the decomposition of the microkernel into multiples managers.

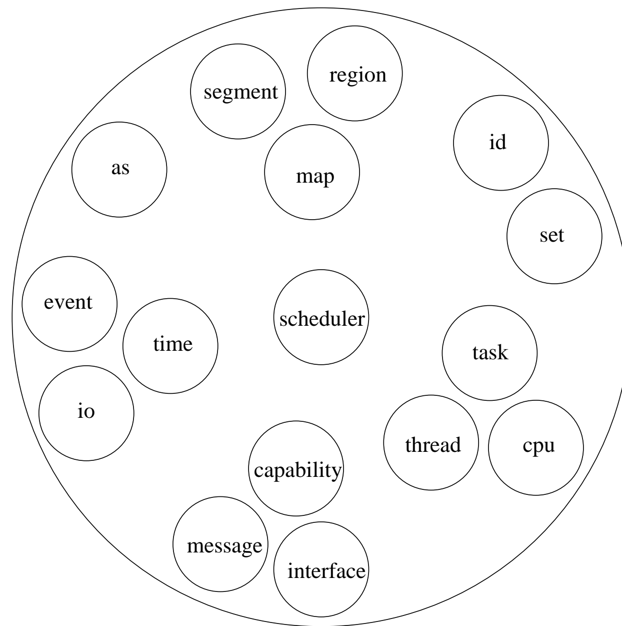


Figure 4.1: kaneton managers.

Note that this decomposition has no direct relation with the *Object-Oriented Programming* paradigm. Indeed, even if kaneton designers tried to reduce the dependencies between managers, some managers remain intrusive as they access the data structures of one or more other managers.

A kaneton object represents a kernel entity. Every object is identified by an unique identifier issued and managed by the **id** manager.

kaneton people believe kernels are composed of data structures and processing, each of these should be very clearly dissociated. Thus, the **set** manager was introduced so that the other kaneton managers use it for storing data without taking care of how it is technically done. The result of such a concept is that the kaneton core source code looks crystal clear very much like pseudo code.

Figure 4.2 depicts the set manager composed of a set container which holds descriptor of the actual data structures.

The kaneton microkernel basically provides the few following functionalities:

- Memory Management;

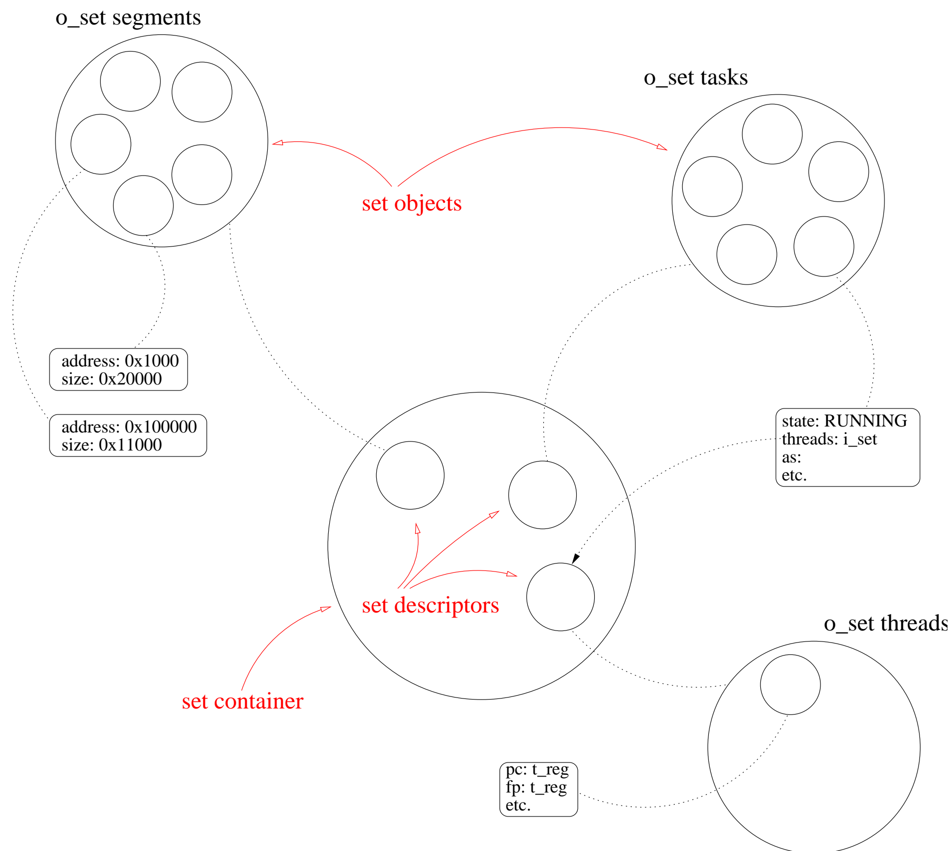


Figure 4.2: kaneton set manager.

- Execution Contexts Management;
- Communication Management;
- Inputs/Outputs Management.

The following details these functionalities through the managers responsible for the related objects.

Memory Management

The memory management consists in providing primitives for performing low-level tasks including reserving, releasing, modifying properties on physical and virtual memory areas. Most kernels divide the memory management into two distinctive parts, one for the physical memory and the other for the virtual one.

In kaneton, there are three managers involved in the memory management.

The first one, called the **address space** or **as** manager manages address space objects. An address space object contains a list of addressable memory locations including physical memory locations and virtual memory locations.

The second manager involved in the memory management is called the **segment** manager. This manager manages the physical memory areas also known as segment objects.

The latter is the **region** manager and is equivalent to the virtual memory manager of other kernels. This manager basically manages virtual memory areas called regions and the mappings between virtual and physical memory locations.

Additionally, the **map** manager provides functionalities for performing both tasks on the segment and region managers, providing a more user-friendly abstraction.

Execution Contexts Management

The execution contexts management consits in providing a complete interface for creating, extending, destroying *etc.* execution contexts.

The **task** manager manages task objects. A task, in kaneton terms, is exactly an execution context. A task object is basically composed of an address space and one or more threads.

The second manager involved in execution contexts management is the **thread** manager since its role is to manage threads. In kaneton, the active scheduled entity is the thread whilst the task is an abstraction of an entire execution context.

In addition, the **cpu** manager is responsible for handling multiprocessor-specific contexts.

Note that another manager, the **scheduler**, takes care of scheduling the threads for execution.

Communication Management

Communication is an important issue since very first microkernels had poor performances due to bad communication design and implementation.

The kaneton **message** manager provides mechanisms for enabling tasks to communicate with each other through different techniques.

The **capability** manager takes responsibility for securing objects sent over a possibly insecure network through cryptographic techniques.

Finally, the **interface** manager represents the point where system calls are received and processed.

Inputs/Outputs Management

In kaneton, every input/output is abstracted in an event through the **event** manager.

kaneton also provides direct device communication through the **io** manager.

4.2 Layers

The kaneton microkernel was designed to be ported on many different - existing or not - architectures. Therefore, the microkernel is divided into two major components: the **core** and the **machine**. The core represents the kaneton code that is independent from the underlying

computer. On the contrary, the machine component contains the code related to the underlying specific hardware.

Since a microprocessor architecture can be used on different mother boards with various chipsets, the machine component is also divided into a **platform** which represents the board package; and an **architecture** which represents the microprocessor.

Note that the behaviour of the **machine** component can change depending on the *platform/architecture* coupling. Therefore, the **glue** component was introduced to deal with the multiple *platform/architecture* combinations.

For more information on the portability system, please refer to *Chapter 6*.

The **library** component provides basic functions required by the kernel very much like the *C Library* provides basic functionalities to the userland applications.

The **modules** component was introduced to attach specific, sometimes machine-dependent, functionalities to the kernel such as a *GDB - GNU Debugger* server, a console or even a *test* server. Those modules are conditionally and statically linked to the kernel.

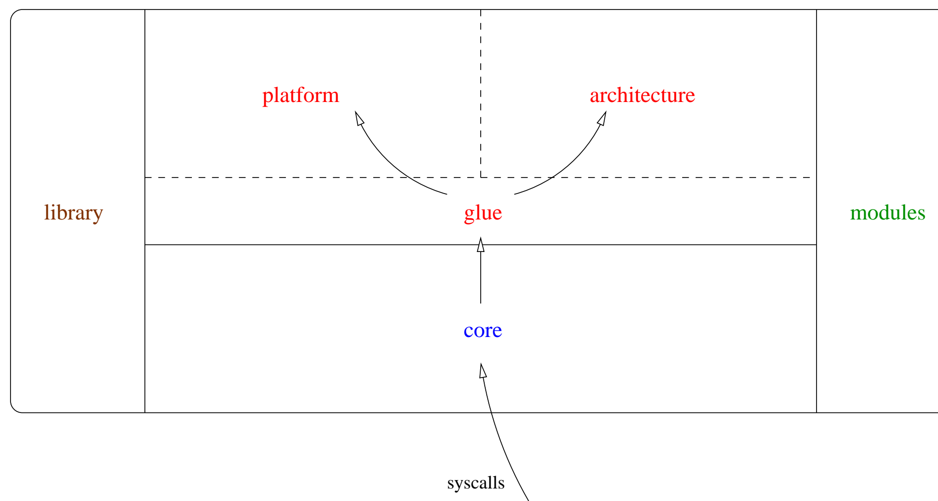


Figure 4.3: kaneton layers.

Figure 4.3 illustrates this decomposition where system calls are first processed by the *core* which then calls the *glue* component. This component knows how to handle operations depending on the current couple *platform/architecture*. It then re-distributes the calls to either the *platform* or *architecture* components.

Chapter 5

Implementation

This chapter specifies some general rules enabling the reader to explore the kaneton implementation without any difficulty.

5.1 Interface

As explained earlier, the kaneton microkernel is subdivided into several managers. Each manager provides an interface to manipulate the kaneton object it is in charge of or something the manager provides an abstraction for.

The naming scheme used for these provided functions is normalised and detailed below.

Manager

Every manager must provide functions for initialisation and cleanup.

Therefore, the `initialize()` function initialises a manager given parameters though, most of the time, the manager does not take any parameter.

The `clean()` function, cleans the manager from anything so that it can return to a stable and clean state.

Information

The following functions are provided for displaying information on either the manager, its objets or both.

The function `show()` displays information on a given identified object.

On the other hand, the `dump()` function displays information on the manager, including a dump of every object held by the manager.

Object

The function `reserve()` allocates an object given some properties whilst the `release()` function releases it.

The function `clone()` creates a copy of the given object. Cloning an object implies cloning every object this object holds or depends on.

The `object()` function is used for retrieving a kaneton object given its identifier. Note that this function is private to the manager though it is sometimes used by other intrusive managers.

Attributes

The following functions enables the caller to access an object attribute.

The function `give()` gives the ownership to another entity whilst the function `flush()` releases every object previously reserved.

In addition, the kaneton managers generally provide functions for modifying a property of a given identified object. Such functions typically take the name of the property to modify.

Finally, every manager provides an `get()` function which returns the state of an object's property while `set()` enables the caller to update a property.

5.2 Organisation

The `kaneton/` directory is organised as follows:

```
core/
include/
library/
machine/
modules/
```

The `library/` directory contains the microkernel-specific C library whilst the `include/` directory contains links to the include directories of every component.

The `core/` directory obviously contains everything related to the core including the source code and header files. This directory contains an `include/` directory with the core header files and a directory for every manager.

The `machine/` directory contains three subdirectories, one for each machine-related component:

```
architecture/
glue/
platform/
```

The `glue/` directory contains subdirectories for every couple platform/architecture supported. Each of these directories contains the source code files, an `include/` directory for the header files and a `layout/` directory which contains information used by the linker in order to place the different elements at precise memory locations.

The `platform/` and `architecture/` directories are organised the same way with subdirectories for every supported platforms and architectures, respectively.

Finally the `modules/` directory contains conditionally-linked static modules *i.e.* pieces of code that add a functionality to the microkernel without being intrusive.

Chapter 6

Portability

In this chapter we will describe the kaneton portability system.

6.1 Background

The kaneton portability system is very different from what other kernels rely on to ensure good portability of the kernel on many architectures.

kaneton people believe that portability systems can be classified into one of the two following categories:

1. The first type of portability system consists in letting the machine-dependent source code defining for example the whole memory manager for its precise architecture.

   If the kernel is ported on another architecture, then the whole memory manager must be rewritten.

   This kind of system enables kernel developers to fully optimize every data structure to better fit every architecture features and needs. On the other hand, if a bug is detected in a generic part of the memory manager, then every machine-dependent implementation must be corrected.

   The problem of such a portability system is the code redundancy in each architecture implementation of the memory manager.

2. The second type of portability system consists in the definition of a machine interface. This machine interface defines common machine-specific basic operations like flushing the *Translation Lookaside Buffer*, installing a new virtual address space *etc.*

   Indeed, these basic operations are present on about every modern architectures. Nevertheless, some architectures—currently existing or not—may possibly not fit this interface.

6.2 kaneton

The two previous portability concepts are actually used in some kernels but kaneton people were not satisfied by those.

Therefore, they decided to design a specific kaneton portability system perfectly fitting in the kaneton requirements based on the kaneton microkernel design. Indeed, the kaneton microkernel is composed of objects on which the kernel applies modifications according to system calls.

Since the objects form the kernel state, the machine-dependent code should act on these objects. Designers decided that the machine-dependent code should be called every time a kaneton object is either created, modified or destroyed.

The kaneton portability system is therefore based on an interface. However, this interface, rather than being well established on well-known architectures functionalities, follows the implicit interface dictated by the kaneton managers.

In other words, rather than calling the machine-dependent code to create a new page directory, the as manager will call the machine-dependent code each time an address space object is created. Therefore, the machine-dependent code might be able to set up an appropriate page directory each time an address space object is reserved.

Having said that the objects represented the kernel state, the machine-dependent code must be able to include machine-dependent data into these objects. For instance, a thread object, from the core point of view, does not contain the registers necessary for storing the state of the current

execution context. The machine-dependent code must therefore include some fields in the thread object structure.

This portability system is experimental and needs more experiments to be validated.

Theoritically, this portability system seems to be far better than the others as there is no code redundancy and there are no constraints for the machine-dependent code.

6.3 Interface

The kaneton portability system relies on c-prepreccesor macros and macro-functions as it enables the machine-dependent source code to *link* itself with the core without anything more.

Therefore, everything is transparent as the core functions use a macro-function and do not know what is called.

Note that these macro-functions are generally handled by the glue component as it is the most suitable for knowing what to do when dealing with portability depending on the underlying platform and architecture.

`machine_include`(`manager`)

> This macro-function can be used by the core to include machine-dependent code for the precise manager `manager`.
>
> More precisely, this enables the machine-dependent code to declare machine-dependent variables in the core.

`machine_data`(`object`)

> This macro-function can be used by the machine-dependent code to add extra fields into the kaneton object type `object`.

`machine_call`(`manager`, `function`, `args...`)

> This macro-function can be used by the core to call the machine-dependent function `function` of the manager `manager`.

6.4 Machine

As said before, the glue component is responsible for setting the links between the core and the platform and architecture components through the portability macro-functions.

Below is presented a common example of how to use these macros in the thread object.

```
typedef struct
{
  t_prior                        prior;

  machine_data(o_thread);
}                                o_thread;
```

In this example, the thread object structure is only composed of a priority attribute which is not sufficient. The architecture probably needs to adds fields for storing the registers state i.e the context.

In order to do that, the glue component has to define the `machine_data()` macro-function:

```
#define         machine_data(_object_)                                  \
  machine_data_##_object_()
```

This way, every call to `machine_data()` will be redirected to another specific macro-function for the given object type *i.e.* `o_thread`.

Then, for each object type, a macro-function must be defined. The following illustrates the case of the `o_thread` object type.

```
#define         machine_data_o_thread()                                \
  struct                                                               \
  {                                                                    \
    t_vaddr             interrupt_stack;                               \
                                                                       \
    union                                                              \
    {                                                                  \
      t_x87_state       x87;                                           \
      t_sse_state       sse;                                           \
    }                   u;                                             \
  }                     machine;
```

On that architecture, the developers decided to store the thread context on a specific stack at the `interrupt_stack` address.

Therefore, once the initial call to the `machine_data()` macro-function is processed, the `o_thread` structure will look like:

```
typedef struct
{
  t_prior                       prior;

  struct
  {
    t_vaddr             interrupt_stack;

    union
    {
      t_x87_state       x87;
      t_sse_state       sse;
    }                   u;
  }                     machine;
}                               o_thread;
```

Finally, the core will behave normally, not knowing that additional fields have been added by the machine-related components whilst the architecture component for instance will be able to access its machine-specific fields without any difficulty.

Chapter 7

Boot

This chapter contains the kaneton boot specifications. Every bootloader willing to run a kaneton microkernel instance needs to comply to these specifications.

The kaneton microkernel is not directly launched when the computer is turned on. Indeed, a **bootloader** first set up an execution environment so that the kernel can be launched properly. The bootloader takes some **inputs** which represents additional files: configuration files, execution files *etc.* For example, the first input the bootloader uses is the kaneton microkernel binary file which is loaded and launched by the bootloader itself. In addition, the second input must be a valid kaneton server since the kaneton microkernel will launch it. As such, the kaneton microkernel assumes the second input is the **system** server, the very first and fundamental server the operating system needs.

The kaneton microkernel is launched with an **init** structure as argument. This structure is described next.

```
typedef struct
{
  t_paddr                        mem;
  t_psize                        memsz;

  t_paddr                        kcode;
  t_psize                        kcodesz;

  t_paddr                        scode;
  t_psize                        scodesz;
  t_vaddr                        slocation;
  t_vaddr                        sentry;

  t_paddr                        init;
  t_psize                        initsz;

  t_inputs*                      inputs;
  t_psize                        inputssz;

  t_uint32                       nsegments;
  s_segment*                     segments;
  t_psize                        segmentssz;

  t_uint32                       nregions;
  s_region*                      regions;
  t_psize                        regionssz;

  t_uint32                       ncpus;
  s_cpu*                         cpus;
  t_psize                        cpussz;
  i_cpu                          bsp;

  t_paddr                        kstack;
  t_psize                        kstacksz;

  t_paddr                        alloc;
  t_psize                        allocsz;

  machine_data(init);
}                                t_init;
```

This structure informs the kernel about the memory layout *i.e.* the location of the different elements in memory as these locations vary according to the machine.

Note that size fields must be aligned on `PAGESZ`. Indeed, the core memory managers behave at the byte level. It is the machine responsibility to call the core with properly aligned sizes.

7.1 Core

Memory

The `mem` and `memsz` fileds specify the offset and the size of the underlying hardware's RAM.

The `mem` attribute is very likely to be set to zero but could vary on specific platforms.

Kernel Code

The two fields `kcode` and `kcodesz` specify the physical memory location and size of the kernel code.

system Server

The *system* server is the very first server launched by the kaneton microkernel. This server is responsible for creating and starting the other servers.

Fields `scode` and `scodesz` specify the location of the physical memory area containing the `system` service code.

`slocation` contains the virtual memory address the code area must be mapped whilst `sentry` contains the virtual address of the code's entry point.

These fields were introduced so that the parsing of the *system* binary is performed by the bootloader, not the kernel. This way, the kernel does not have to take care of handling multiples executable file formats such as *ELF*, *COFF etc.*

Indeed, the kernel receives the *init* structure, creates a new task, maps the *system* service code and points the task's thread to the entry point.

init Structure

The `init` and `initsz` fields contain the location and size of the *init* structure itself.

Inputs

Inputs are additional files passed to the *mod* service.

Theses files are gathered together in a single memory area specified through the `input` and `inputsz` fields.

This area first contains metadata with the `t_inputs` structure:

```
typedef struct
{
  t_uint32                        ninputs;
}                                 t_inputs;
```

The `ninputs` field of the metadata obviously indicates the number of inputs. These inputs follow the metadata as explained next.

Inputs are actually organised in an array of elements composed of the input metadata and the input contents. The input metadata is described by the `t_init` structure:

```
typedef struct
{
  char*                          name;
  t_psize                        size;
}                                t_input;
```

Everything related to inputs is packed in a single location so that passing these information to the *mod* service is as simple as passing the address and size of this memory area *i.e.* `inputs` and `inputssz`.

Segments

Segments passed by the bootloader to the kernel indicate the zones of physical memory which are already used. Thus, the kaneton microkernel can initialise its memory managers, especially the segment manager, according to those zones.

The `nsegments` attribute indicates the number of segments in the array located in the memory area specified by `segments` and `segmentssz`.

Elements of the array of segment are of the following type:

```
typedef struct
{
  t_paddr                        address;
  t_psize                        size;
  t_perms                        perms;
}                                s_segment;
```

Regions

Regions provided through the *init* structure indicate the kernel which memory locations are already mapped.

The kernel can use these information for initialising its memory managers, in this case the region manager, so that data structures are coherent.

As for the segments, the `nregions` regions are gathered in an array located at `regions` of size `regionssz`.

Every element of the region array are of the following type:

```
typedef struct
{
  t_uint32                       segment;

  t_vaddr                        address;
  t_paddr                        offset;
  t_vsize                        size;
  t_opts                         opts;
}                                s_region;
```

Note that the `segment` field of this last structure correspond to an index in the array of segments.

Processors

The `ncpus` field indicates the number of cpu elements in the array located at `cpus` of size `cpussz`. Each element is of the following form:

```
typedef struct
{
  i_cpu                          cpuid;
}                                s_cpu;
```

Additionally, the `bsp` field indicates the identifier of the boot processor.

Kernel Stack

The kernel stack is specified through the `kstack` and `kstacksz`.

Allocator's Pre-Reserved Memory

The memory area specified by `alloc` and `allocsz` is used by the kaneton microkernel for performing allocations in order to set up the memory managers.

Indeed, when the kaneton microkernel starts, the memory managers are not initialised and hence cannot provide memory management functionalities. Traditional kernels tend to rely on a specifically designed physical memory manager for this purpose. This design leads to an ugly implementation.

kaneton people wanted to avoid that and decided to rely on a pre-reserved memory area provided by the bootloader.

7.2 Machine

The reader would have probably noticed the use of the `machine_data()` macro-function in the *init* structure.

Indeed, the *init* structure can include machine-dependent information that will be later used by the kaneton machine components.

For instance, the *Intel Architecture 32-bit* kaneton bootloader includes, through the `machine_data()` macro-function, the following information in the *init* structure:

```
#define         machine_data_init()                                  \
  struct                                                             \
  {                                                                  \
    t_ia32_gdt                  gdt;                                 \
    t_ia32_directory            pd;                                  \
  }
```

Chapter 8

Licenses

In this chapter we will details the kaneton-related licenses.

The kaneton project might be considered as an open-project since source code is provided.

Nevertheless this is not the case as this project is used as material for operating system courses.

Therefore, people implementing the kaneton microkernel should not make their source code available. To avoid problems, especially students cheating, kaneton people decided to use a kaneton-specific license forbidding source code distribution.

The *kaneton license* is based on a more generic license, the *pedagogical licence*.

The next sections will contain these licenses' descriptions.

8.1 Pedagogical License

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

2. You must not copy or distribute copies of the Program's source code, object code or executable form without explicit authorization from the maintainers.

   If you have this authorization, you must conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, provided that you meet all of these conditions :

   (a) You must not publish your work without explicit authorization from the maintainers.

   (b) You must send to the maintainers any work that in whole or in part contains or is derived from the Program or any part thereof.

   (c) You must cause any work that you send, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole under the terms of this License.

   (d) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (e) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

4. Access to the Program's source is granted if either:

(a) You want to make the Program evolve

(b) You have pedagogical goals

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. We may publish revised and/or new versions of this License from time to time. It may evolve considering new contributors needs. Contact us if you have any request. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission, we sometimes make exceptions for this.

11. THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS

IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

8.2 kaneton License

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This licence is nothing more than a link to the pedagogical licence.
2. Any program under the kaneton licence is in fact under the terms and conditions of the pedagogical licence.