The kaneton microkernel :: development

kaneton people

February 17, 2011

This document describes how people can contribute to the kaneton microkernel project and the rules they have to follow.

This document must be read by everyone contributing to the kaneton microkernel research project implementation.

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

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

Contents

1 Introduction **5**

2 Source Tree **7**

3 Community **13**

4 Rules **17**

5 Tools **21**
5.1 Internal . . . 22
5.1.1 Environment . . . 22
5.1.2 Configure . . . 34
5.1.3 View . . . 36
5.1.4 Export . . . 37
5.1.5 Transcript . . . 40
5.1.6 Cheat . . . 40
5.1.7 Test . . . 41
5.1.8 Prototypes . . . 51
5.1.9 Control Panel . . . 53
5.2 External . . . 55
5.2.1 Mailing-List . . . 55
5.2.2 Repository . . . 56
5.2.3 Wiki . . . 57
5.2.4 Project Management . . . 58

6 Languages **59**
6.1 Make . . . 60
6.1.1 Naming . . . 60
6.1.2 Environment . . . 60
6.1.3 Layout . . . 60
6.1.4 Style . . . 61
6.2 Python . . . 62
6.2.1 Naming . . . 62
6.2.2 Environment . . . 63
6.2.3 Layout . . . 63
6.2.4 Style . . . 64
6.3 Assembly . . . 64
6.3.1 Inline Assembly . . . 64
6.3.2 Naming . . . 65

6.3.3 Layout . . . 65
6.4 C . . . 66
6.4.1 Naming . . . 67
6.4.2 Layout . . . 70
6.4.3 Style . . . 76
6.4.4 Control Structures . . . 79
6.4.5 kaneton . . . 81
6.5 LaTeX . . . 83
6.5.1 Naming . . . 83
6.5.2 Layout . . . 83
6.5.3 Style . . . 85

7 People **91**
7.1 Project . . . 92
7.2 Tools . . . 92

8 Licenses **93**
8.1 Pedagogical License . . . 94
8.2 kaneton License . . . 96

Chapter 1

Introduction

In this chapter, the kaneton microkernel project is briefly introduced in order to emphasize some of its characteristics that makes contribution specific in several ways.

kaneton is a educational purpose microkernel project. This project aims at providing a very clear, commented and maintainable microkernel source code in order to allow people interested in operating systems internals to look at the source code and understand it very quickly.

The kaneton project is basically composed of the source code of the microkernel itself, scripts to perform complex tasks and various documents from design papers to lecture materials.

The most important thing to remember is that the whole project is intended to be understood as well as possibly maintained by everyone. As a result, contributions must comply with the level of clarity expected by the project.

These rules are discussed in this paper in order to inform every new contributor of what makes a good contribution.

The remaining of this document is organised as follows. *Chapter 2* introduced the kaneton project organisation through the source code hierarchy. Next, *Chapter 3* describes how a contributor should behave in a development community. *Chapter 4* introduces the general rules which apply to any context around the kaneton project. Then, the tools inherent to the kaneton project are listed in *Chapter 5* with some guidelines about how to use them properly. *Chapter 6* explicitly describes languages rules informing the developer of the coding style to respect. *Chapter 7* draws a list of the people in charge for the different parts and tools of the project. Finally, *Chapter 8* contains information about the licenses related to the kaneton microkernel project.

Chapter 2

Source Tree

In this chapter we will briefly describe the kaneton microkernel project source tree.

The kaneton microkernel reference source tree looks like the following listing:

```
boot/
cheat/
configure/
environment/
export/
history/
kaneton/
license/
sample/
test/
tool/
transcript/
view/
```

boot/

The `boot/` directory contains the component related to the kaneton boot process including the *bootstrap* and especially the *bootloader*.

The bootstrap is kept for the kaneton educational project while the bootloader is actually used for setting up the execution context before launching the kaneton microkernel.

cheat/

Since the kaneton microkernel is implemented by students, the kaneton people need to check whether students are cheating by re-using parts of previous years projects or other kernel source codes available on the *Internet*.

To avoid cheating, kaneton people developed a software checking for commonalities between different source codes.

This directory contains scripts that performs these verifications. However, the students work over the years are not stored in this directory but in the `history/` directory instead.

configure/

This directory contains everything necessary for configuring its own kaneton microkernel development environment through the compiling process to the boot system.

Any new contributor should first look at this directory. However, note that this directory mainly contains tools targeting final-users rather than kaneton contributors. Indeed, for instance, the *configure* utility aims at providing a user-friendly way for configuration but does not take advantage of the power of the kaneton development environment.

Contributors should then learn about how the development environment works while final-users should use the *configure* tool.

environment/

This directory contains everything necessary to the kaneton development environment.

The kaneton development environment allows different developers to interact on the development of the same microkernel in a pretty easy way.

The development environment aims at providing developers to possibility to work in a collaborative manner without interfering with each other. These developers are likely to run different operating systems on different microprocessors. In addition, the kaneton microkernel can be targeted for different microprocessor architectures. The development environment was introduced to cope with these combinations by providing profiles, each profile describing the behaviour of a component: underlying operating system, target architecture, user-specific stuff *etc.*

As a result, each developer can use a different operating system and microprocessor architecture with its own specific compiling flags, kaneton parameters *etc.* without modifying another developer's configuration.

The development environment is detailed in *Section 5.1.1.*

export/

The `export/` directory contains scripts used to generate a kaneton tarball in order to be distributed to the students at the beginning of the kaneton educational project.

Indeed, these scripts rearrange the kaneton hierarchy hidding some important directories the students do not need to be aware of. Moreover some source code parts are removed since the students have to rewrite these pieces of code as assignments.

These scripts are also used for making backups and distribution tarbalss of the kaneton microkernel.

history/

The `history/` directory contains the students work over the years in the universities and schools the kaneton project was used as an operating system course's implementation material.

The tools of the `cheat/` directory use these students works for performing cheating verifications.

kaneton/

This directory is the most important of the project since it contains the whole microkernel source code.

The directory is composed of three important subdirectories: `core/`, `machine/` and `include/`. These subdirectories are described next.

include/

This directory is the unique include point of the kaneton microkernel.

This directory contains symbolic links to the `include/` directories for the *machine*, *glue*, *platform* and *architecture* components.

core/

This directory contains the kaneton core source code. The *core* represents the machine-independent source code/

The directory is divided as shown below:

```
as/
region/
scheduler/
segment/
set/
task/
thread/
[...]
```

Each directory represents a kaneton core manager. For more information on the kaneton core, please refer to the appropriate document: *The kaneton microkernel :: core*

machine/

This directory contains the machine-dependent source code.

The machine is composed of three components, the *plateform* which represents the board supporting the devices: microprocessors, memories, peripherals *etc.*; the *architecture* which represents the microprocessor architecture and finally the *glue* which assemble these two components forming the *machine*.

For more information on the kaneton portability system, please refer to the *The kaneton microkernel* book.

library/

This directory contains whatever the kernel needs *i.e.* very much like the *C* library for userland applications.

modules/

This directory contains additional functionalities that can be statically added to the kernel.

For example, the test system is composed of a kernel part which, when activated, is compiled and linked with the kernel.

license/

This directory contains the licenses used for any program or document in relation with the kaneton microkernel project. Indeed, the kaneton microkernel is under the *kaneton license* which is described in depth in the documents contained in this directory. Note that these licenses are also available in *Chapter 8*.

Each student has to read and agree with the kaneton license before implementing or even using the kaneton microkernel project..

Indeed, every user of the kaneton-related stuff is considered as having implicitly accepted the kaneton license.

sample/

This directory contains demonstration servers.

test/

Since the kaneton microkernel is used as a material for operating system courses, the kaneton microkernel reference, which is the basis of students work, must be extremely reliable.

The kaneton project therefore contains a set of tools in order to validate the kaneton reference implementation behaviour. These tools are also used for evaluating the correctness of the students implementation.

The `test/` directory contains the set of kaneton scripts and tests for validating a kaneton microkernel implementation.

tool/

This directory contains additional scripts and configuration files used by the kaneton development environment or the kaneton developers.

As examples, this directory contains scripts for generating prototypes, building a boot device *etc.*

transcript/

This directory contains real-time recorded sessions. These sessions can be replayed in order to present a feature of the development environment or of the kaneton microkernel.

view/

This directory contains all the kaneton documents including kaneton administrative documents, examinations, lectures materials, kaneton papers and books *etc.*

Additionally, scripts are provided in order to very easily build and display these documents.

Chapter 3

Community

This chapter discusses what is a community and how contributors must integrate the kaneton community.

kaneton can obviously be considered as an open source community although the produced soure code is actually not open source.

Driving an open source community is complicated since people have different personal goals at working on a free project. Some people contribute for the knowledge, other for building the next generation system, other to provide free open source softwares, other to become famous *etc.*

kaneton is a community driven microkernel that acts with the best interest of the students at heart. Rules and regulations that keep the project moving forward are fundamental even if the size of the kaneton community is relatively small, for now.

Indeed, the main objective of the kaneton project remains to be as understandable as possible in order to lead students to implement parts of it very quickly.

The remaining of this chapter draws a list of rules contributors must agree to respect.

Objective

kaneton aims at providing a powerful, understandable and maintainable microkernel. This objective must be kept in mind of every contributor since many design and implementation were/are/will be made according to this precise objective.

Note that the kaneton microkernel does not intend to be a desktop operating system nor an as optimised as the Linux operating system. Every contributor should be well-aware of that in order to avoid behaviours stating that a feature is fundamental or useless for performance concerns, for instance.

This rule does not prohibit people to suggest ideas but instead regulates behaviours of people who wants to change major design and/or implementation choices for bad reasons.

Behaviour

Open source projects does not mean constraint-free projects. The kaneton people, whilst being relatively young, try to act for the project's good by behaving remarkably in the kaneton community.

Therefore, contributors are asked to do the same by avoiding some bad/young behaviours.

1. Follow the rules. People who do not respect these rules could be banned from the kaneton project.
2. Avoid the *cowboy* behaviour consisting for a contributor to implement a feature without discussing about its usefulness with the community first. Another effect of this behaviour can be to distract the contributors from its major focuses.
3. Always act and think in the project interest rather than your personal interest.
4. Respect the other kaneton people, especially the ones who have worked on this project for a long time and who made this whole project possible. When people disagree, they are asked to do it respectfully.
5. Take your responsability when you realise that you did something wrong: insults, mistakes in an implemented feature *etc.*

6. *"The Perfect is the Enemy of the Good!"*: even nice contributors can unintentionally do bad things by being perfectionists and/or too much into the project and/or obsessed with process.

7. ... *Politness*, *Respect*, *Trust* and *Humility* are the key qualities that make a good contributor in any community.

Communication

The communication mainly takes two forms in the kaneton microkernel project: the *mailing-list* for internal communication and the *kaneton website* for external communication. The *Developers Intranet* is another source of communication as well as the commit logs *etc.*

The rules related to these tools are described in *Chapter 5* and will therefore not be discussed here.

Every contributor must take the time to communicate in the mailing-list as well as through the public website. Indeed, kaneton people must, frequently, briefly describe what they are working on in order to inform the other contributors who are not aware of everyone's current work. Note that these kind of messages are very different from messages generated by repository commits. Indeed, while these commit messages indicate a modification, they do not describe the whole work behind them.

Additionally, contributors can communicate informing the kaneton community of their unavailability for the next two months, for instance. This behaviour allows people to be aware that some tasks will not be done because the contributors in charge of it cannot work at this moment.

Althoug people are highly welcomed to communicate, some rules apply to avoid further problems.

First, any new contributor should obviously read the kaneton documents before asking anything which has already been discussed and decided, unless the developer knows exactly what he/she is talking about. Indeed, asking too many questions about the source code is a form of disrespect to the other contributors. Moreover, many things can be found out just looking at the kaneton documents and/or source code.

Although people are asked to communicate, people are also asked to act respectully. Contributors should not respond to every message in every discussion, this is a ridiculous behaviour. Instead, every developer should carefully read the discussion, think about its response and then write a clear message stating his point of view, ideas *etc.*

Depending on the contributor status, reading the mailing-list frequently is absolutely fundamental as some people rely on other contributors' decisions, advices *etc.*

Finally, the mailing-list must be considered as the official internal communication medium. If contributors previously had a private conversation, in real-life or on *IRC* for instance, the discussion must be reported on the mailing-list so that everyone can take these new ideas into account.

Work

Working on the kaneton microkernel project does not imply low-level programming all the time. Indeed, the kaneton project is composed of two parts: the kaneton microkernel research project and the kaneton educational project.

Although the microkernel research project requires highly skilled programmers, it also needs documentations and some tools for performing important tasks as diverse as generating the prototypes, testing the microkernel behaviour, generating the documentation *etc.*

The educational project essentially needs documentation, lecture materials and tools for managing the project: testing the students' implementation, checking if some students cheated and many others.

This means that kaneton people must contribute to every type of task that need to be done. Also, contributors are asked to well document any work they have done including source code comments but also through kaneton official documents which are then made available on the website.

Supervisor

A supervisor is attached to each new contributor for a certain, not fixed, period of time. The role of this supervisor is to advise, correct and encourage the newcomer so that it integrates well the kaneton community.

The supervisor will be someone having a well understanding of the new contributor's project. A high level of communication is expected between the supervisor and the contributor. A direct phone communication is highly recommanded through *Skype* for instance.

Trust

A new contributor joining the kaneton project must acquire the trust of the community. Therefore, the contributor first does not get any access to restricted tools and must submit patches to its supervisor who review them, taking care of advising the newcomer of its mistakes.

At the end of the test period, the community decides whether the contributor is helpful to the project or not. Then, the contributor is either granted of full access to kaneton tools or eliminated from the project.

Chapter 4

Rules

This chapter describes the rules which apply to the whole kaneton microkernel project. These rules can therefore be considered as the most important ones.

kaneton

The most important thing when contributing or talking about a project is to know how to write its name.

The authors of the project decided kaneton must be written in lower case: **kaneton**. The same goes for the kaneton project names which contain a *k* letter in lower-case: *k1*, *k2*, *k3* and so on.

Also remember that the kaneton project is composed of two distincts sub-projects named, the *kaneton microkernel educational project* and the *kaneton microkernel research project*.

Header

Files must start with a file header. This file header specifies the project name, the license of this file, the file name, the author and date of the file creation and finally the author and date of the last edition. This header must comply to the following template, depending on the sequence of characters used for comments. The example below illustrates the template for *C* files.

```
/*
 * ------- header -------------------------------------
 *
 * project       <project>
 *
 * license       <license>
 *
 * file          <file location>
 *
 * created       <first author>   [<creation date>]
 * updated       <last author>   [<last update date>]
 */
```

Then, an example for LaTeX files:

```
%
% ------- header -------------------------------------
%
% project       kaneton
%
% license       kaneton
%
% file          /home/mycure/kaneton/view/book/development/community.tex
%
% created       julien quintard   [mon may 28 19:44:49 2007]
% updated       julien quintard   [mon may 28 19:48:07 2007]
%
```

Note that the kaneton project provides an *Emacs* file which contains everything necessary to build such headers. This file is located on the developers' private *Wiki a.k.a.* the *Developers Intranet*.

Obviously, the project and license fields must be filled, in the kaneton project context, with *kaneton* and *kaneton*, respectively. The author field must contain the author's full name - firsname and lastname - in lower-case letters. Note that auto-generated values must comply to the general kaneton rules especially they must be in lower-case letters and must not exceed 80 characters in width.

Markings

Any developer must put the sequence `xxx` everywhere a piece of code is considered as unfinished. This way, any unfixed piece of code can be easily retrieved via a very simple command line or script.

Naming

When using a language which does not support namespaces, the developer should prefix every entity by the package, module *etc.* name it actually belongs to.

As long as it is possible, entities must be named with a *unique* word, excluding the namespace prefix.

Names must obviously be expressed in English, lower-case letters and without any spelling mistake.

Composite names should be separated by a dash - when the language allows it, including file names. Otherwise, the underscore character _ must be used.

Layout

Files are composed of sections in order to make the organisation clearer. Each section starts with a specific header and then contains code, text *etc.* related to the section.

A section header is basically a commented separator.

Any file must include a *header* section as explained above. Moreover, some sections are mandatory depending of the type of file. For instance, even configuration files, description files, frame file, *Python* files *etc.* must provide an *information* section.

Below is an example of a *Python* script which illustrates the use of sections:

```
#
# ------- header -------------------------------------
#
# project       kaneton
#
# license       kaneton
#
# file          /home/mycure/kaneton/foo/foo.py
#
# created       julien quintard   [sun may 13 11:04:52 2007]
# updated       julien quintard   [mon may 28 12:42:57 2007]
#

#
# ------- information --------------------------------
#
# this script is used to illustrates the use of sections in kaneton
# files.
#

#
# ------- imports ------------------------------------
#

import env
```

```
import sys
import re

#
# ------- globals ---------------------------------------
#

g_string = 'kaneton'

#
# ------- functions -------------------------------------
#

#
# main()
#
# this function does the main work: displaying a string.
#
def main():
  env.display(env.HEADER_NONE, g_string, env.OPTION_NONE)

#
# ------- entry point -----------------------------------
#

if __name__ == "__main__":
  main()
```

Files related to any language should provide a section for the core content like *rules* for *Make* files or *functions* for many other languages.

For more information on the mandatory sections, please refer to the sections about the language your are interested in and/or look at examples in the kaneton repository.

However, note that a section which does not contain any code must not appear in the file.

Files must not exceed 80 characters in width, including the trailing newline character. Moreover, the *DOS CR+LF* line terminator must not be used. Finally, there must not be any whitespace at the end of a line.

An indentation a two spaces must be used. Moreover, the *Emacs* default indentation must always be taken as a referencial.

Rules

Any contributor which notices a misuage of these rules must inform the kaneton community and especially its supervisor so that the mistake can be corrected.

Moreover if something is missing in this book, any contributor is welcomed to inform the community about it so that a rule is added or modified.

Contributors are asked to read the rules enumerated in this book but also to look at the kaneton repository as it contains many examples of applications of these rules.

Chapter 5

Tools

This chapter describes every tool kaneton contributors use on a regular-basis.

5.1 Internal

The kaneton project contains several tools which makes the developer life easier. This section describes these tools in order for the contributor to use them but also to understand them well.

5.1.1 Environment

Over the years, the kaneton microkernel evolved, starting with a very simple introduction to low-level programming and finally to a complete microkernel development.

kaneton people wanted to lead students to a complete microkernel development to finally introduce distributed computing. This would not have been possible if students had to build an entire development environment because developing such an environment is a whole project by itself.

As a result, kaneton people decided to provide students a complete development environment. The kaneton development environment is composed of make files, python scripts and configuration files. This development environment can be considered as one of the major kaneton tools since contributors use it everytime.

The kaneton development environment aims at providing an easy and portable way for managing the kaneton microkernel project from a development point of view. Therefore, the kaneton environment provides everything necessary for compiling, assembling, *etc.* These tasks highly rely on the underlying running operating system as well as on the kaneton microkernel's target microprocessor. Moreover, the user could need to redefine some behaviours depending on its personal operating system configuration to use a specific C compiler for instance.

The kaneton development environment provides a layered organisation of profiles, each profile defining variables and functions used by the final environment engine. The goal of the layered model is to allow layers to override the definitions of lower-level layers.

Profiles

The development environment is composed of profiles including a *host* profile which describes the behaviour of the underlying operating system, a *kaneton* profile which parameters the kaneton microkernel and a *user* profile which permits the user to redefine lower-level layers' declarations.

These profiles eventually hold sub-profiles which define variables and functions. These actual profiles are accessed according to user-defined shell variables.

Host

The *host* profile essentially describes how to perform basic tasks: compile, assemble, change the current directory, display a message *etc.* These tasks rely on the operating system currently running as well as on the target processor which kaneton will be built for. For these reasons, there are several host sub-profiles.

Let us suppose a developer is running a *Linux* operating system and that kaneton will be built for running on a *PowerPC* microprocessor. In such a case, the C compiler program will be different depending on the microprocessor *Linux* is running on. Indeed, if Linux is running on a *PowerPC* microprocessor, then using the default compiler should produce *PowerPC* object files. This is well-known to be the common compiling way. On the other hand, if *Linux* is running on a different microprocessor, then a cross-compiler must be used to produce binary objects targeting a specific different microprocessor architecture, the *PowerPC* architecture in our example.

To avoid this issue, a *host* sub-profile name is composed of two parts separated by a slash. The first part is the name of the operating system and the latter is a pair source/target processors separated by a period. For example, *linux/ia32.ppc* names a *host* profile representing a *Linux* operating system on a *Intel 32-bit* microprocessor which aims at building a kaneton microkernel for a *PowerPC* target architecture. Needless to say that *linux/ia32.ia32* represents a non cross-compiling environment.

To avoid configuration duplications, it is common to see the configuration file of a host sub-profile to include files of the parent directory as shown below:

```
linux/
  linux.desc
  linux.conf
  linux.mk
  linux.py
  ia32.ia32/
    educational -> .
    optimised -> .
    smp -> .
    ia32.desc
    ia32.conf
    ia32.mk
    ia32.py
  ia32.mips64/
    mips64.desc
    mips64.conf
    mips64.mk
    mips64.py
```

Note that the files `linux.*` are not directly included by the development environment engine since *linux* is not a valid host profile name.

Two host profiles are illustrated here. The first one is named *linux/ia32.ia32* while the second's name is *linux/ia32.mips64*.

For example, the *linux/ia32.mips64 host* profile represents a *Linux* operating system running on a *Intel 32-bit* microprocessor while kaneton is built for a *MIPS 64-bit* target architecture. This profile is likely to include the `linux.*` files of the parent directory since there are not much difference between all the *linux/*.* host* profiles. However, such a profile will certainly redefine the binary paths of the C compiler, linker *etc.* in order to produce *MIPS 64-bit* binary objects.

To conclude, the *host* sub-profile is accessed by the following construct:

```
profile/host/${KANETON_HOST}/${KANETON_ARCHITECTURE}
```

With, for instance, the following values:

```
KANETON_HOST = linux/ia32
KANETON_ARCHTECTURE = ia32/educational
```

Note that the possibility to include files in the configuration syntax allows very similar profiles to share a huge amount of definitions.

Boot

The *boot* profile is both used for configuring the boot components such as the *loader etc.* but also to set up the way a bootable system is built.

kaneton

The *kaneton* profile is composed of four sub-profiles: *core*, *machine*, *library* and *modules* for the *kaneton* microkernel sub-components respectively. Likewise, *machine* is sub-divided into *platform*, *architecture* and *glue*.

The *core* sub-profile contains variables for parameterizing the kaneton core. The *platform* and *architecture* sub-profiles focus on the configuration of the platform- and architecture-dependent code of the kaneton microkernel, respectively.

The user-defined shell variables `${KANETON_PLATFORM}` and `${KANETON_ARCHITECTURE}` are used to address the *platform* and *architecture* sub-profiles, respectively.

User

Let us suppose that a developer would like the kaneton microkernel to use a specific memory management entirely based on a *Slab Allocator* and with all microprocessor optimisations enabled. These user-specific configurations are actually allowed by the *user* profile.

The user-defined shell variable `${KANETON_USER}` defines the name of the *user* profile. This profile contains user-specific configurations allowing a contributor to overwrite lower-level layer defintions in order to specialise a behaviour.

The kaneton project also provides a tool allowing users to configure their development environment. This tool is named *configure* and is available from the kaneton project root directory. For more information about this tool, please refer to *Section 5.1.2*.

Requirements

The whole kaneton development environment needs exactly two fundamental tools to work. The first one is *GNU make*, used to build powerful make files, and the second one is *Python*, used to write portable scripts. If an operating system has these two tools, then kaneton can certainly be developed on it.

As said previously, the user has to specify some shell variables which are used by the kaneton development environment engine. These variables are described below:

- `${KANETON_USER}`: the name of the kaneton developer.

  A *user* profile name must be composed of the first name, a period and finally, the last name of the developer.
- `${KANETON_HOST}`: the name of the host which is composed of a couple operating system/microprocessor.
- `${KANETON_PYTHON}`: the path of the python binary.

  This path is required since the very first scripts which set up the configured environment are written in *Python*.
- `${KANETON_PLATFORM}`: the name of the target platform.
- `${KANETON_ARCHITECTURE}`: the name of the target architecture.

Note that once the configured environment is set up, these variables are no longer used by the kaneton environment engine. Indeed, instead, the kaneton environment operations are based on the *host* profile on which rely the configured environment.

The profiles names must all be in lowercase letters. Below are some examples of what could contain these variables:

```
KANETON_USER='julien.quintard'

KANETON_HOST='linux/ppc'
KANETON_HOST='windows~cygwin/ia32'

KANETON_PYTHON='/usr/bin/python'

KANETON_PLATFORM='ibm-pc'
KANETON_PLATFORM='sgi/o2'
KANETON_PLATFORM='sgi/octane'

KANETON_ARCHITECTURE='mips64'
KANETON_ARCHITECTURE='ia32/educational'
KANETON_ARCHITECTURE='ia32/smp'
```

Organisation

The development environment configuration files and scripts are located in the *environment/* directory. The directory contains the three following scripts:

```
critical.py
initialize.py
clean.py
```

The `critical.py` script essentially generates a configured development environment. The result of this generation are two files called `env.mk` and `env.py` which contains the configured environment variables and functions for the *Make* files and *Python* scripts, respectively. This file is called critical because it does not rely on the portable development environment as it generates it.

The `initialize.py` script relies on the file `env.py` previously generated. This script set up everything necessary for building the kaneton microkernel based on the configured environment.

Finally, the `clean.py` script cleans everything installed by the `initialize.py` script and removes the generated configured environment files.

The generation of the configured environment is done by going through the configuration files of all the profiles and sub-profiles associated to the user configuration. In other words, the kaneton environment engine processes the configuration files according to the layered organisation described below, starting with the lowest-level layer thourgh the highest one.

```
profile/
profile/host
profile/host/${KANETON_HOST}.${KANETON_ARCHITECTURE}
profile/boot
profile/boot/${KANETON_PLATFORM}.${KANETON_ARCHITECTURE}
profile/kaneton
profile/kaneton/core
profile/kaneton/machine
profile/kaneton/machine/platform
profile/kaneton/machine/platform/${KANETON_PLATFORM}
profile/kaneton/machine/architecture
profile/kaneton/machine/architecture/${KANETON_ARCHITECTURE}
profile/kaneton/machine/glue
profile/kaneton/machine/glue/${KANETON_PLATFORM}.${KANETON_ARCHITECTURE}
profile/kaneton/library
profile/kaneton/modules
profile/user
profile/user/${KANETON_USER}
```

In this layered organisation, a variable defined in, for instance, the *host* profile could be overwritten anywhere in the upper-level layers `kaneton/`, `kaneton/architecture/${KANETON_ARCHITECTURE}/`, `user/` *etc.*

The *host* and *kaneton* profiles are theoretically completed separated. However, the environment engine does not check for such unauthorised overridings. Therefore the *core* configuration could override a variable previously defined in the *host* profile.

Finally, the *user* profile can override any definition to adjust the environment to its needs.

The environment engine looks for the following types of file in the kaneton environment profile directories:

- `.conf`: the *configuration* files contains variable definitions. These files are gathered by the development environment engine for generating the configured environment files.
- `.desc`: these *description* files contain descriptions of the variables of the current profile or sub-profile. These descriptions are used by the *configure* tool.
- `.mk`: the *Make* files usually contain the implementation of the kaneton *Make* interface.
- `.py`: the *Python* files usually contain the implementation of the kaneton *Python* interface.

The engine supposes that there is no variable or function overriding in a single profile. More precisely, if there are more than a single configuration file in a directory, the engine cannot guarantee anything on the order these files will be processed. As a result, the overridings could differ depending on the processing order.

Besides, although the environment engine gathers every *configuration* files it finds in the environment profiles directory, it is highly recommended to provide a single *configuration* file per profile directory. This file should be named according to the name of its profile. For more details, take a look at the environment directory which contains existing profiles.

Moreover, the *configure* tool requires the *user* profile to contain a single *configuration* file named, as explained above, `${KANETON_USER}.conf`.

The kaneton development environment engine first gathers the *configuration* files and process them creating an in-memory list of configuration variables. Then it generates the configured environment files `env.mk` and `env.py`. Indeed, the engine outputs the configuration variables in each file and then append the content of the *Make* files and *Python* files of the profiles to the configured environment files `env.mk` and `env.py`, respectively.

Note that a special rule is included in `env.mk` so that the configured files are regenerated if the environment engine detects that an environment file has been changed since the last initialization.

Syntaxes

Description

The *description* files describe the environment variables in order to specify what kind of value a variable can take *etc.*

The description syntax is based on the *YAML* language.

Examples of variable descriptions named `_FOO_`, `_BAR_` and `_CHICHE_` are given next:

```
#
# _FOO_
#
- variable: _FOO_
  string: the foo flag
  type: set
  values:
    Off: -D_FOO_FLAG_=0
    On: -D_FOO_FLAG_=1
  description: |
    This is a description of a two-state variable _FOO_.

#
# _BAR_
#
- variable: _BAR_
  string: the bar parameter
  type: set
  values:
    Simple: ${BAR_SIMPLE}
    Normal: ${BAR_NORMAL}
    Optimised: ${BAR_OPTIMISED}
  description: |
    This is another parameter which can take three values: simple,
    normal and optimised.

#
# _CHICHE_
#
- variable: _CHICHE_
  string: the most powerful optimisation
  type: any
  description: This is the magic kaneton optimisation.
```

Note that the environment engine never takes these descriptions into account. Indeed, this is the role of the *configure* tool.

In this syntax, variables are classified according to the type of value they can take: `set` and `any`.

A `set` variable can take any value in a given list of values. In this case, the `values` field contains couples string/value the variable can take. The `string` is displayed by the *configure* tool while the `value` is assigned to the variable.

Finally, a `any` variable represents a variable which can take any value.

The `string` fields were introduced to avoid displaying internal non user-friendly variable names and/or values. Therefore, the *configure* tool will always display literal `strings` rather than variable names or values which are likely to do not make any sense to the user.

Configuration

The *configuration* files contains the actual variable definitions through a very simple syntax.

The syntax allows both assignments and completion as shown in the next example:

```
FOO = bar
FOO += baz
FOO = kaneton
```

The `FOO` variable first takes the initial value `bar`. Then, the value `baz` is added to the previous `FOO`'s value leading the the value `bar baz`. Finally, the last assignment overwrites the previous definition by setting the value of `FOO`'s variable to `kaneton`.

The configuration syntax enables variable references. These references can be both environment variable or shell variable. The following example illustrates this.

```
BAR = ${FOO} is a very powerful microkernel
SH = the shell currently used is $(SHELL)
```

The reader certainly notice the `${}` construct is used to reference a kaneton environment variable while the `$()` one references a shell variable.

Finally, a configuration file can also include another file using the `include` statement:

```
include ../an/other/file/far/../far/../away
```

This construct is very useful to centralize definitions common to multiple profiles or sub-profiles in a single location.

Note that kaneton environment variables start and end with an underscore to avoid naming collisions. Another solution would have been to use a prefix `KANETON_` as it is stipulated in the general kaneton rules but this would have led to very long names.

Make

The *Make* files must implement the whole kaneton *Make* interface which will be described next.

The syntax used in these files is based on the *GNU Make* syntax.

Python

The *Python* files must implement the whole kaneton *Python* interface.

The syntax used in these files is based on the *Python* syntax.

Interfaces

Make

In this section we will detail the make interface that every host profile must implement. The reader should look closer to the host profiles already implemented.

Since the *GNU Make* syntax does not provide any name space feature, every kaneton *Make* function is prefixed by `env_` in order to avoid name conflicts.

Note that the *Make* development environment must take care of setting the `PYTHONPATH` shell environment variable with a value including the `_PYTHON_INCLUDE_DIR_` kaneton environment variable so that scripts can use kaneton *Python* packages.

`env_perform`(`command`)

> This function performs an action according to the given `command` argument.
>
> Additionally, if the `_OUTPUT_` environment variable is set to `${_OUTPUT_VERBOSE_}`, the function displays the command on the output before performing it.

`env_display`(`color`, `action`, `file`, `indentation`, `options`)

This function displays a message representing an action performed by the kaneton *Make* interface.

The option `$(OPTION_NO_NEWLINE)` can be used not to output the trailing newline.

`env_cd`(`directory`, `options`)

This function changes the current working directory.

`env_pull`(`file`, `options`)

This function returns the content of the `file`.

`env_launch`(`file`, `arguments`, `options`)

This function launches a new program/script/make *etc.*

This function must look at the file name in order to determine how to launch it. Moreover, the function must move the the directory where is located the file before launching it.

`env_preprocess`(`preprocessed file`, `c file`, `options`)

This function launches the C preprocessor the `c file` and generates the `preprocessed file`.

`env_compile-c`(`object file`, `c file`, `options`)

This function compiles a `c file` generating an `object file`.

`env_lex-l`(`c file`, `lex file`, `options`)

This function generates a `c file` from a `lex file`.

`env_assemble-S`(`object file`, `S file`, `options`)

This function assembles an `S file`.

`env_static-library`(`static library file name`, `object files and/or libraries`, `options`)

This function builds a static library from object files.

`env_dynamic-library`(`dynamic library file name`, `object files and/or libraries`, `options`)

This function builds a dynamic library from `object files and/or libraries`.

`env_executable`(`executable file name`, `object files and/or libraries`, `layout file`, `options`)

This function builds a executable from object files and/or libraries. The `layout file` describes where to place the different data: code, read-only data, stack *etc.*

The option `$(ENV_OPTION_NO_STANDARD)` tells the function not to use the operating system standard stuff: libraries, includes *etc.*

`env_archive`(`archive file name`, `object files`, `options`)

This function builds an archive of objects from multiple `object files`.

`env_remove`(`files`, `options`)

This function removes the files in the list.

`env_purge`()

This function just cleans the current working directory from unecessary files.

`env_prototypes`(`files`, `options`)

This function generates prototypes in relation with the given `files`.

`env_headers`(`files`, `options`)

This function generates header dependencies for the `files` by building a *Make* dependency file named `${_DEPENDENCY_MK_}`.

The generated files `${_DEPENDENCY_MK_}` are removed by the environment engine when cleaning the configured development environment.

`env_version`(`file`)

This function generates a version `file` from the operating system's informations: user, host, date *etc.*

`env_link`(`link`, `file`, `options`)

This function creates a `link` to the `file`.

`env_compile-tex`(`file`, `options`)

This function compiles the file `file`.tex and generates a readable document.

`env_document`(`file`, `options`)

This function builds a document by calling the `env_compile-tex()` function.

The option `$(ENV_OPTION_PRIVATE)` configures the document by setting the LaTeX definition `\mode` to the value `private`. This option was introduced to deal with documents which contain information which must be kept private to the students.

Note that a temporary file named `${_DEPENDENCY_TEX_}` is created by this functions storing the LaTeX definition `\mode`. The developer should take care of removing this file in the `clear` *Make* file rule.

`env_view`(`file`, `options`)

This function launches a viewer for the readable document generated by the function `env_compile-tex()`.

Note that file, as for the `env_compile-tex()` function, does not have any filename suffix.

Python

In this section we will detail the kaneton *Python* interface that every *host* profile must implement.

The *Python* language was designed in a portable way. For this reason, the major part of the *Python* interface is implemented by the *host* generic profile.

Note that the *Python* language provides modularity through packages. Therefore, each *Python* script has to import the `env` package generated by the development environment engine: `environment/env.py`. Then, environment functions and variables are accessed through this package.

Below are described the functions implemented by the `env` package.

Note that the *Python* development environment must take care of setting the `PYTHONPATH` shell environment variable with a value including the `_PYTHON_INCLUDE_DIR_` kaneton environment variable so that scripts can use kaneton *Python* packages.

`display`(`header`, `text`, `options`)

This function outputs some text to the screen depending on the `header`: `HEADER_NONE`, `HEADER_OK`, `HEADER_ERROR` or `HEADER_INTERACTIVE`.

`pull`(`file`, `options`)

This function returns the content of the `file`.

`push`(`file`, `content`, `options`)

This function writes the `content` in the `file`.

`temporary`(`options`)

This function creates a temporary file system object.

The options `OPTION_FILE` and `OPTION_DIRECTORY` specify which type of object to create.

`cwd`(`options`)

This function returns the path of the current working directory.

`input`(`options`)

This function waits for an input from the user.

`launch`(`file`, `arguments`, `options`)

This function launches a new program/script/make file *etc.*

This function must look at the file name in order to determine how to launch it. Moreover, the function must move the the directory where is located the file before launching it.

The option `OPTION_QUIET` makes the `launch()` function do not print anything on the output screen.

`copy`(`source`, `destination`, `options`)

This function copies the file `source` to `destination`.

`link`(`source`, `destination`, `options`)

This function builds a link between the file `source` and the file `destination`.

`remove`(`target`, `options`)

This function removes the `target` which can be either a file or a directory.

`list`(`directory`, `options`)

This function lists the file system objects contains in the `directory`.

The options `OPTION_FILE` and `OPTION_DIRECTORY` specify which type of object to list.

`cd`(`directory`, `options`)

This function changes the current working directory to `directory`.

`search`(`directory`, `pattern`, `options`)

This function looks for files matching the given `pattern`.

The options `OPTION_FILE` and `OPTION_DIRECTORY` specify which type of object to list while the `OPTION_RECURSIVE` option tells the function to explore the whole file system sub-tree.

`pack`(`directory`, `file`, `options`)

This function makes an archive `file` of the `directory`.

`unpack`(`file`, `directory`, `options`)

This function extracts the archive `file` into the `directory`, if specified.

`mkdir`(`directory`, `options`)

This function builds a new directory named `directory`.

`load`(`file`, `device`, `path`, `options`)

This function copies the `file` on the specificed `device`, more precisely at the location `path`. The device can be virtual: an image.

The options `OPTION_DEVICE` and `OPTION_IMAGE` specify on which type of device the file must be copied.

`stamp`(`options`)

This function returns a current date.

`record`(`transcript`, `options`)

This function starts recording a session and transcripts it into `transcript`.

`play`(`transcript`, `options`)

This function plays a previously recorded `transcript`.

`locate`(`file`, `options`)

This function tries to locate the program `file` on the system.

`path`(`path`, `options`)

This function returns information on the given `path`.

The options `OPTION_FILE` and `OPTION_DIRECTORY` specify which information the caller is interested in. The option `OPTION_EXIST` indicates whether the `path` object exists or not.

`info`(`options`)

This function returns information on the system.

The option `OPTION_CURRENT_DIRECTORY` returns the sequence of characters used for accessing the current directory.

5.1.2 Configure

The *configure* tool provides the final user a very user-friendly software for customizing its development environment.

Recall the development environment is basically composed of three profiles: *host* which describes the operating system behaviour, *kaneton* which parameterizes the kaneton microkernel and *user* which contains some user-specific definitions.

The kaneton development environment is thus used to configure the environment behaviour as well as the kaneton microkernel itself.

The `environment/` directory, and more precisely the environment profile directories, contain *description* files which actually describe the environment variables. These files are not used by the development environment but rather by the *configure* tool.

The `configure/` directory is composed of *frame* files which contain frame descriptions. A frame can be seen as a menu presented to the final user. A frame is composed of meta-data but also sub-frame and variable entries.

The *configure* tool works as follow. It starts by processing the environment development configuration files as the environment engine did for the generation of the configured environment files. Note that the *configure* tool also processes the description files. Also, it focuses on variables and actually ignores the *Make* and *Python* functions.

Once this step is done, the tool gets a list of configured and fully described variables. Then, the *configure* tool displays the first frame and waits for the user to choose an entry.

The user has the possibility to either move to another menu - if any sub-frame entry is present - or configure a variable of the list. If the user chooses to configure a variable, then, the tool displays information based on the variable's description.

Every modifications of the development environment are private to the actual user. Therefore, any variable modification adds or modifies an entry in the related *user* profile's configuration file.

Note that the *configure* tool is not an environment configuration files editor. Indeed, this tool targets final users and therefore tries to be as simple as possible.

The basic *configure* behaviour consists in displaying the final variable's value. If the user enters a new value, no matter whether there is a relation with its previous value, the tool creates/modifies an entry in the *user* profile's configuration file overriding any previous definition.

For instance, consider the `_FOO_` development environment variable and the following configuration files:

In `environment/profile/environment.conf`:

```
_FOO_                   =               initial
```

In `environment/profile/core/core.conf`:

```
_FOO_                   +=              addon
```

Let us suppose the user enters the following value instead of the current one: `initial addon`.

```
_FOO_                   =               something new
```

Then, the *configure* tool creates a new entry into the *user* profile configuration file:

```
_FOO_                           =                       something new
```

Finally, note that when the *configure* tool is launched, it first tries to detect whether the user is a newcomer or not. If it is, then the tool asks the user to create a new *user* profile, step by step. These actions are performed in the `critical.py` script of the `configure/` directory.

Requirements

The *configure* tool relies on the *Dialog* software which is present on many *Unix* systems. Indeed, the *configure* tool is a user-friendly configuration utility.

Since *configure* needs to update the *user* profile configuration file, this file must be unique and easy to locate. Therefore, the *configure* tool supposes this configuration file is accessible at:

```
environment/profile/user/${KANETON_USER}/${KANETON_USER}.conf
```

Finally, the *Python* module *PyYAML* is required as the description and frame files use the *YAML* syntax, as described next.

Syntax

The syntax of the *frame* files `.frm` is based on *YAML*.

As said previously, a frame is composed of sub-frame and variable entries. A sub-frame entry contains a name and a path to the sub-frame *frame* file while a variable entry only contains the name of the variable. This variable name is then used to retrieve the variable description.

In addition, a section containing a title and a description is used to customize the display presented to the user.

The example below illustrates this very simple syntax:

```
#
# ------- information ---------------------------------
#
# this is the main menu of segment manager.
#

#
# ------- general -------------------------------------
#

- title: Segment Manager
  description: |
    This section contains configuration about the core segment manager.

#
# ------- frames --------------------------------------
#

- frame: Optimisations
  path: subsections/optimisations.desc

- frame: Machine-dependent
  path: subsections/machine.desc

#
```

```
# ------- variables ----------------------------------
#

- variable: _FOO_

- variable: _BAR_

- variable: _CHICHE_
```

5.1.3 View

The *view* tool serves as a document database as well as a tool for building and displaying documents in an easy way.

The kaneton documents are classified, each directory corresponding to a class of documents. Below are listed the subdirectories of the `view/` directory.

```
bibliography/
book/
exam/
feedback/
figures/
internship/
lecture/
logo/
package/
paper/
talk/
template/
```

The `template/` directory contains templates for every class of document. The `bibliography/` and `logo/` directories contain, obviously, the bibliography which is common to all the documents, and the logos, respectively. The `figures/` directory contains figures common to all the documents while the `package/` directory contains additional LaTeX packages.

The directories `curriculum/`, `exam/` and `feedback/` contain documents in relation with teaching. The `curriculum/` directory contains documents such as the educational project year planning *etc.* The `feedback/` directory contains documents which are distributed to the students at the end of the kaneton project in order to get feedback for improving the project for the next years. Needless to say the `exam/` directory contains everything related to examinations while the `talk/` directory contains conference talks and various presentations of the education project for instance.

The other directories contain the actual kaneton documentation. The *books* represent the main documents targeting any public: contributors, teachers, students *etc.* The *papers* are lighter documents intended to present a specific feature, design *etc.* The *lectures* are the courses materials, generally composed of presentation slides. Finally, the *internship* documentation is composed of documents written by people partially involved in the kaneton project.

Any document is composed of a *Make* file and one or more LaTeX files. The *Make* file always has the same form with little variations depending on the type of document. For more information on the rules applying to the *Make* and LaTeX files, please refer to their respective sections: *Section 6.1* and *Section 6.2*.

The *view* tool basically starts looking for `.tex` files and builds a list of directories containing documents. Then, it provides to the user the possibility to build and display a given document. If no document name is given on the command line, then the tool draws a list of the available documents.

People contributing to the kaneton documents must take care of following the rules in relation with the LaTeX language. Moreover, contributors should look at the existing documents to understand to logic behind all these rules.

Finally, note that nobody should create a document without discussing it on the mailing-list first. Especially, be very careful in naming your documents as people took good care of this directory in order to avoid it to become messy.

If a document already exists with the same name, then go through the mailing-list in order to decide whether to keep the current version. If people decide to keep a document, then, the contributor in charge of writing the new one should re-organise the documents by creating archives for each year.

5.1.4 Export

The *export* tool was introduced for making the releasing process easier. The tool does its job based on action description YAML files.

Recall the kaneton microkernel project is used as a material for operating system courses. The source code of the microkernel is distributed to the students with some parts missing. Then, students have to re-write these pieces of code in order to prove their well-understanding of the kernel internals. Additionnaly, the kaneton project is also a research project in operating systems design.

As a result, the *export* tool sometimes has to build a release with pieces of code removed, sometimes not. The tool has been build to be flexible enough to fulfil multiple needs.

The tool is build on a modular approach. Each module provides an action that can be used within a YAML file.

A YAML file describes the sequence of actions to be performed to make an export. All the operations are performed in a temporary directory displayed at the begin of the export.

Here is the list of the currently available modules :

- svnexport This module takes no argument. It performs an svn export of the kaneton SVN repository.
- localexport This module takes no argument. It performs a copy of the user's working copy.
- import This is a built-in module, it takes one argument : *filename*. This argument is the name of another YAML file to import and run. The YAML file has to be in the export folder, no path and no file extension should be put in the argument.
- initenv This module takes no argument. It de-initializes the kaneton environment. It can be used after calling localexport to clean the environment before making a snapshot.
- fremovepattern This module deletes every file or directory that matches the *pattern* argument which is a regular expression. For instance, to clean the export of all the subversion folders, the pattern `.*\.svn` can be used.
- fremove This module deletes the file or directory described in the argument *path*. It should be a relative path based on the repository root directory, like for instance : `boot/bootloader`.
- fmove This module moves the file or directory described in the argument *src* to the path described in the argument *dst*. If the entity to be moved is a file, the *dst* argument must be a filename, not a directory where the file should be moved.

- freplace This module copies the file described in the argument *src* to the path described in the argument *dst*. The *dst* argument must be a filename, not a directory where the file should be copied. This module can be used to replace files by templates stored in `export/data`.
- symlink This modules creates a symbolic link named by the argument *name* pointing to the file described in the argument *target*.
- bremove This module removes a block of lines in a file. It takes one argument : *id* which has this syntax : `path/to/file::block_name`. The syntax of blocks declaration is explained below.
- breplaceThis module replaces a block of lines in a file. It takes two arguments : *id* which has this syntax : `path/to/file::block_name` and *data* which is the text to be put instead of the removed lines. The syntax of blocks declaration is explained below.

Blocks Syntax

As explained previously, pieces of code are being identified by blocks, in order to be processed by the export tool.

The code below illustrates how to tag some code to make a block:

```
/*                                                      [block::clone] */

/*
 * this function clones a segment.
 *
 * steps:
 *
 * 1) get the original segment object.
 * 2) reserve a new segment of same size with same permissions.
 * 3) copy the data from the old segment.
 * 4) call machine-dependent code.
 */

t_error                 segment_clone(i_as                      asid,
                                      i_segment                 old,
                                      i_segment*                new)
{
  o_segment*            from;
  t_perms               perms;

  SEGMENT_ENTER(segment);

  /*
   * 1)
   */

  if (segment_get(old, &from) != ERROR_NONE)
    SEGMENT_LEAVE(segment, ERROR_UNKNOWN);

  [...]

  /*
   * 4)
   */

  if (machine_call(segment, segment_clone, asid, old, new) != ERROR_NONE)
    SEGMENT_LEAVE(segment, ERROR_UNKNOWN);

  SEGMENT_LEAVE(segment, ERROR_NONE);
}

/*                                                   [endblock::clone] */
```

The markings at the top `[block::clone]` and bottom `[endblock::clone]` of this example indicate the *export* tool that the block called *clone* in this file contains the function segment_clone. This block can then be used in YAML files to perform actions such as removal or replacement on the content of this block.

Note that the marked areas must not overlap, the *export* tool's behaviour being undetermined in such cases.

YAML Syntax An export is described in a YAML file. It contains a list of modules to call with their arguments. The tag for the module to call is *operation*. Some optional tags must be added, depending on the module that is being called.

Here is an example of a YAML file :

```
--
-
  operation: localexport

-
  operation: initenv

-
  operation: fremovepattern
  pattern: .*\.svn

-
  operation: import
  filename: k1

-
  operation: fremove
  path: tool/ctc

-
  operation: fmove
  src: kaneton/libc
  dst: libc

-
  operation: freplace
  src: export/data/Makefile
  dst: Makefile

-
  operation: breplace
  id: kaneton/core/id/id.c::test
  data: |
    if (a)
      b = 2;
    else
      c = 3;

-
  operation: bremove
  id: kaneton/core/core.c::foo

-
  operation: symlink
  name: kaneton/core/test
  target: kaneton/core/region
```

All the paths that are used must be relative to the repository root directory.

Usage

All YAML files must be in the `export/` directory.

They can be called by running : `make export-filename` where filename is the name of a YAML file without the extension.

It's important that the YAML file that is run starts by one of the two export commands : `svnexport` or `localexport`. For that reason, all YAML files are not executable, since some of them are being imported by others.

Note that the commands are executed within the temporary directory sequentially. If your YAML script moves a folder, all other commands following the move command dealing with files in that folder must refer to the new location of the folder within the temporary folder.

You have to write a YAML file for each export need you have, but operations can be factorized by using the import command.

The export tool doesn't make a tarball for the moment. Instead, it displays the temporary folder when run, and does all the work in that folder, so the user can then make a tarball of this folder, or work in this folder directly.

5.1.5 Transcript

The `transcript/` directory is composed of two tools related to the management of transcripts. The *record* tool captures a shell session while the *play* tool replays a captured session.

These tool were introduced to allow students to make a dynamic presentation of their kaneton implementation's features and possibilities. These dynamic presentations were supposed to replace the oral examinations.

These transcripts are not used by the main contributors of the kaneton project yet. However, any teacher interested by this tool can use it.

The `transcript/` directory contains subdirectories which classify the transcripts.

The only transcript class currently in place is named *basic* and contains transcripts illustrating the use of the kaneton internal tools.

Note that the *Unix host* profiles rely on the well-known *Unix script* software. Moreover, a tool is provided in `tool/script/` for replaying *script* captured sessions.

5.1.6 Cheat

The *cheat* tool checks whether students cheated by using pieces of code from other students' implementation of the current and/or previous years.

The `history/` directory is composed of directories organizing the kaneton students implementations over the years and for every school and university the education project has been used for. Then, each subdirectory represents a year and contains subdirectories for each students group of this year.

Each student group directory contains a `sources/` subdirectory containing the snapshots of the different kaneton stages: *k0*, *k1*, *k2* and so on.

The *cheat* tool takes a school, a year and a stage as arguments. Its first task is to generate the fingerprints of the other kaneton implementations.

Once the fingerprints have been generated, they are gathered into a database file. The tool then performs the verification process by comparing snapshots against each others.

Note that in order to prevent the tool from detecting matches in the source code that has been provided by the teachers, the tool first removes the parts common to both the students snapshots and the base snapshot. Additionally, to reduce the amount of work to be done, everything contained in the `_CHEAT_FILTER_` environment variable is removed from the students' snapshots. This variable is likely to contain directories such as `environment/`, `license/`, `tool/` *etc.*

Finally, the tool generates a *HTML* page summarising the matches found between the students. The matches are classified according to the number of tokens found.

Note that teachers are asked to add student snapshots to the repository in a careful way, taking care that snapshots do not contain any object file, revision control directories such as `.svn/` *etc.* and that once extracted, the snapshot produced a single `kaneton/` directory. Moreover, if the student snapshot does not follow the base organisation, false-positive matches will emerge.

The *cheat* tool is based on another tool whose name cannot be revealed here. For more information, please contact your supervisor.

5.1.7 Test

The *test* tool enables students to test their kaneton implementation against a set of tests that have been designed by the contributors. Below is briefly described the terminology used by this tool in order to give the reader an overview of the general scenario involving students, the administrator and the server running the test system.

- A *certificate* is used to make sure clients can authenticate the test server;
- Every certificate is sealed by a cryptographic *key*;
- Each user is provided with a *capability* in order to identify herself to the server;
- These capabilities are sealed by a *code* which the server uses in order to detect illegally forged capabilities;
- A *configuration* specifies the number of tests a user is allowed to requests the server;
- The user's *database* is generated based on a configuration and maintains the current user's state on the server including the number of tests performed so far, the kaneton implementations submitted for evaluation *etc.*
- A *snapshot* is a kaneton implementation in its shipping form;
- The *machine* represents the target *platform/architecture* couple on which a snapshot is supposed to be tested or evaluated for instance;
- An *image* represents a kaneton snapshot compiled in a bootable form;
- A *test* is a function included in the kernel which performs a specific set of operations and possibly prints information to the console;
- The tests are often gathered together in a *suite* which represents the testing unit students are offered to trigger for their kaneton implementation;

- Once a snapshot is received by the server in order to be tested, the system compiles it into an image. The server also takes care to include the tests in the compilation process so that they can be triggered. These pre-compiled tests are referred to as the *bundle*;
- The image can then be tested by triggering the tests of the suite. The image is therefore booted in an emulated *environment*. This environment can sometimes be chosen and offers a trade-off between simplicity and realism. The most common environments are *QEMU* and *Xen*;
- Depending on the success of the tests, a set of results is generated and compiled in a *bulletin* file;
- Finally, the server retrieves this bulletin, adds some meta information such as the date of the test, the environment and machine used *etc.* and stores everything in a *report*. Note that this report is also sent back to the user so she can consult it;
- Students can also decide to submit their kaneton implementation for a specific *stage* for future evaluation. Note that suites and stages are completely different though they often bear the same names: *k0*, *k1*, *k2* etc;
- The administrator can decide to evaluate the snapshots which have been submitted for a stage by invoking a script which will attribute grades according to the *point*s associated with every test.
- Finally, a *statement* is produced containing the grades of every student for a given stage.

The following describes the *test* tool according to the user's role regarding the system: either the administrator who sets up the system or a student who uses it in order to improve and/or evaluate his implementation.

Administrator

The administrator is responsible for setting up the system but also maintaining it on a daily basis.

Requirements

The *test* tool must be installed on a publicly accessible server since the server script will be waiting for incoming requests. Note that by default, the clients assume the test server to be accessible at the address: `https://test.opaak.org:8421`.

Besides, since the purpose of the *test* tool is to run the students' kaneton implementation in emulated environments, both *QEMU* and *Xen* should be available though one might want to configure the tool for supporting a single environment, *QEMU* for instance.

Note that the test system has been developed with *Python 2.6* and may be out of date by the time the administrator sets it up. In addition, the system depends on a variety of *Python* packages including *argparse*, *yaml*, *pyopenssl*, *hmac*, *pickle*, *xmlrpc*, *subprocess* among others.

Finally, the administrator should make sure the following applications are installed since some test scripts need them: *dd*, *mkfs.ext2*, *mount*, *umount*, *mutt* and *mkisofs*.

Set Up

The first step for an administrator consists in generating the necessary files, especially the certificates, code and capabilities required for securing the test system.

The `test/utilities/` directory contains the scripts that perform such operations. Note that all the generated files are stored in the `test/store/` directory.

First the *CA - Certification Authority*'s and server's certificates must be generated. The first is used to issue certificates while the latter is used for clients to identify the server with absolute certainty.

```
$> make certificate
[+] generating the CA and server's key/certificate pair
[+] CA key/certificate generated
[+] server key/certificate generated
[+] CA and server's key/certificate pair generated and stored
$>
```

The next step consists in generating a code for the administrator to issue capabilities but also for the server to verify that the received capabilities have not been illegally forged.

```
$> make code
[+] generating the server's code
[+] server code successfuly generated and stored
$>
```

With a server code, the students' and contributor's capabilities can be built, hence granting them the right the contact the server.

The following generates the contributor's capability. This capability is special in the way that contributors can perform any operation in a completely contrain-free manner.

```
$> make capability-contributor
[+] generating the contributor's capability
[+] contributor's capability generated and stored
$>
```

In contrast, the following command generates a set of capabilities for the students belonging to the school referred to as *"epita::2010"*. Note that the script requires the `history/epita/2010/` to be populated with the groups and their `people` file.

```
$> make capability-school@epita::2010
[+] generating students' capabilities
[+] extracting the students from the history 'epita/2010'
[+] students information retrieved
[+] generating the students' capabilities:
[+]   epita::2010::group11
[+]   epita::2010::group10
[+]   epita::2010::group13
[+]   epita::2010::group12
[+]   epita::2010::group33
[+]   epita::2010::group32
[+]   epita::2010::group17
[+]   epita::2010::group30
[+]   epita::2010::group19
[+]   epita::2010::group18
[+]   epita::2010::group5
[+]   epita::2010::group4
[+]   epita::2010::group7
[+]   epita::2010::group6
[+]   epita::2010::group1
[+]   epita::2010::group3
```

```
[+]   epita::2010::group2
[+]   epita::2010::group15
[+]   epita::2010::group9
[+]   epita::2010::group8
[+]   epita::2010::group14
[+]   epita::2010::group31
[+]   epita::2010::group16
[+]   epita::2010::group24
[+]   epita::2010::group25
[+]   epita::2010::group26
[+]   epita::2010::group27
[+]   epita::2010::group20
[+]   epita::2010::group21
[+]   epita::2010::group22
[+]   epita::2010::group23
[+]   epita::2010::group28
[+]   epita::2010::group29
[+]   epita::2010::group34
[+] students' capabilities generated and stored
$>
```

In addition, the administrator could decide to generate or re-generate a capability for a specific student of a school. The following shows an example for such an action.

```
$> make capability-student@epita::2010::group8
[+] generating the student's capability:
[+]   epita::2010::group8
[+] student's capability generated and stored
$>
```

The next step consists in the databases generation. A database contains the state of a user profile including the number of test requests, the quota for such tests, the submitted snapshots and so forth. The database files are absolutely fundamental to the server since such databases are updated after each client's request.

The syntax for generating databases follows the one for capabilities, as shown next for the contributor.

```
$> make database-contributor
[+] generating database from contributor's configuration
[+] contributor's database generated and stored
$>
```

Once the certificates, code, capabilities and databases generated, the administrator can move on to the deployment process.

Deployment

The deployment basically consists in copying the `test/` environment to the test server though one might want to copy the whole kaneton environment or the smallest subset of the `test/` directory which should, in this case, include the following absolutely necessary items:

- The `test/environments/` directory which contains the descriptions of the supported test environments;
- The `test/images/` directory which contains a script for automatically generating a *Debian Live* system which is used for compiling a kaneton snapshot into a bootable image;

- The `test/packages/` directory which contains the *ktp - Kaneton Test Package* required by the server-side standalone scripts for manipulating files such as databases, capabilities *etc.* but also for performing cryptographic operations and send/receive *XMLRPC* requests;
- The `test/scripts/` directory which contains the fundamental scripts for building bootable images, distributing the capabilities to the students through emails, evaluating the submitted snapshots and so on;
- The `test/server/` directory which contains the server script for handling the clients' requests;
- The `test/stages/` directory which contains the files requirement for evaluating the students' snapshots;
- The `test/store/` directory which contains the generated files such as the users' databases, the server's code and certificate; and
- The `test/suites/` directory which contains the files describing the tests to be including in a given tests suite.

Once copied, the administrator only needs to launch the server script located in the `test/server/` directory, as shown below:

```
$> ./server.py
[meta] serving on 88.191.84.128:8421
```

Note that a few additional steps may be required depending on the current state of the kaneton development.

The first of these steps may consist in generating a *Debian Live* system since this is absolutely required for the test system to work. For more information regarding the generation of such an image, please refer to the `test/images/` directory.

The second step should consist for the administrator in building the kaneton tests bundle. The bundle represents a pre-compiled set of tests that is included in the students' snapshot compilation process. The tests are pre-compiled in order to prevent leaking information since students could very well dump the content of those tests and force the compilation to fail, hence retrieving the source code in the compilation process' error log.

In order to generate such a bundle, the administrator must first activate the *test* module, as show next:

```
_MODULES_               +=              test
```

Then, the administrator must move to the `test/tests/` directory and launch a compilation process through the following command:

```
$> make
```

Once generated, the test bundle, located in `store/bundle/[machine]/` must be copied to the server, at the same location.

Finally, for more information on the server script, please refer to the `test/server/` directory.

Scripts

Although the deployment process is pretty straightforward, the administrator is required to manage the test system through several scripts.

First, the *distribute* script must be used by the administrator to send the capabilities to the respective owners so that the students can use the test system. Note that this script relies on the *Mutt* mailing system for sending the emails containing the attached capabilities.

```
$> ./distribute.py
recipients:
  contributor
$>
```

While the *construct* script enables the administrator to build a bootable image from a kaneton snapshot, the *stress* script takes a bootable image and triggers the tests belonging to the given test suite. Note that both scripts are directly used by the server script for building and testing the received kaneton snapshots.

```
$> ./construct.py -snapshot kaneton.tar.bz2                    \
                  -image kaneton.img                           \
                  -environment xen                             \
the kaneton image has been constructed in 'kaneton.img'
$> ./stress.py -image kaneton.img                              \
               -suite k2                                       \
               -environment xen                                \
               -verbose
segment
  permissions/01 :: true
id
  simple :: true
  clone :: true
  multiple :: true
$>
```

Note that the administrator could also test a kaneton image manually, especially through the following command:

```
$> qemu -fda kaneton.img -curses
```

Besides, note that an administrator willing to include a new test in the system would probably want to test it locally first since testing through the server takes some time. In order to test locally, the administrator must first activate the bundle module in its user profile `environment/profile/user/$KANETON_USER/$KANETON_USER.conf`:

```
_MODULES_               +=              bundle
```

Then, the administrator must trigger the test by calling the test function manually in its kaneton implementation. For instance, in order to trigger the *kaneton/core/task/guest* test, the administrator could add the following line after `kernel_initialize()` and before running the test system in `kaneton/core/core.c`:

```
[...]

module_call(console, message,
            '+', "starting the kernel\n");
```

```
assert(kernel_initialize() == ERROR_OK);

/* XXX[temporary] */
test_core_task_guest();

module_call(test, run);

[...]
```

Once the kaneton image rebuilt, the administrator can boot it locally through *QEMU* and get the output, hence check that the test went as excepted:

```
$> qemu -fda environment/profile/user/${KANETON_USER}/${KANETON_USER}.img
```

Back to the server side, the *evaluate* script can be used by the administrator in order to assign grades to the snapshots submitted by the students. The script generates a statement containing the results of this evaluation process.

```
$> ./evaluate.py -stage k2                                          \
                 -pattern "^epita::2010::.*$"
the statement has been saved in '../store/statement/20101102-223645.db'
$>
```

Finally, the *dump* script takes any *YAML*-based file and displays its inner structure in a hierarchical manner.

```
$> ./dump.py -path ../store/statement/20101102-223848.db
meta:
  reference:              20.0
  stage:                  k2
data:
  epita::2010::group7:
    date:                 2010/11/02 20:46:44
    grade:                16.0
    snapshot:             20101102-204644
    members:
      email:              admin@opaak.org
      name:               admin
    configurations:
      Xen:
        report:           20101102-224213
        notch:            4
        score:            4
      QEMU:
        report:           20101102-223848
        notch:            4
        score:            0
```

Student

The student has the possibility to request actions from the test server through the client script located in `test/client/`.

Requirements

Although the client script is integrated in the kaneton environment, it also makes use of the *ktp*. Therefore, as for the server, the client depends on a variety of *Python* packages including *yaml*, *pyopenssl*, *hmac*, *pickle*, *xmlrpc*, *subprocess* among others.

Use

The client script enables the user to request one of the five operations described below.

```
$> make
[!] usage: client.py [command]

[!] commands:
[!]   information
[!]   submit-[stage]
[!]   test-[environment]::[suite]
[!]   list
[!]   display-[identifier]
[!]   retest-[identifier]
$>
```

The *information* operation requests the server to return information on the current state of the user's profile. The information returned range from the number of tests performed, the quota for every test suite to the available stages or the snapshots having been previously submitted.

```
$> make information
[+] configuration:
[+]   server:                https://test.opaak.org:8421
[+]   capability:            /data/mycure/repositories/kaneton/environment/profile/user/julien.quintard/juli
[+]   platform:              ibm-pc
[+]   architecture:          ia32/educational

[+] information:
[+]   profile:
[+]     identifier:          contributor
[+]     community:           contributors
[+]     members:
[+]       name:              admin
[+]       email:             admin@opaak.org
[+]   suites:
[+]                          k1
[+]                          k3
[+]                          k2
[+]                          kaneton
[+]   stages:
[+]                          k1
[+]                          k2
[+]                          k3
[+]   environments:
[+]                          qemu
[+]                          xen
[+]   database:
[+]     reports:
[+]       xen:
[+]         ibm-pc.ia32/educational:
[+]           k3:
[+]           k2:
[+]           k1:
[+]       qemu:
[+]         ibm-pc.ia32/educational:
[+]           k3:
[+]           k2:
[+]           k1:
[+]     settings:
[+]       xen:
[+]         ibm-pc.ia32/educational:
[+]           k3:
[+]             requests:    0
[+]             quota:       -1
```

```
[+]           k2:
[+]             requests:     0
[+]             quota:        -1
[+]           k1:
[+]             requests:     0
[+]             quota:        -1
[+]       qemu:
[+]         ibm-pc.ia32/educational:
[+]           k3:
[+]             requests:     0
[+]             quota:        -1
[+]           k2:
[+]             requests:     0
[+]             quota:        -1
[+]           k1:
[+]             requests:     0
[+]             quota:        -1
$>
```

The *test* command enables the user to trigger a test suite for the current kaneton implementation on the specified environment such as *QEMU* or *Xen* for instance.

The server then returns the resulted report which the client stores locally in `test/store/report/`. In addition, the client displays a quick summary of the report in order for the user to know whether things went as expected.

```
$> make test-xen::k2
[+] configuration:
[+]   server:                 https://test.opaak.org:8421
[+]   capability:             /data/mycure/repositories/kaneton/environment/profile/user/julien.quintard/juli
[+]   platform:               ibm-pc
[+]   architecture:           ia32/educational

[+] report(20101103:140601):
[+]   segment                                                                 [1/1]
[+]   id                                                                      [3/3]
$>
```

The *list* command enables the user to display the identifiers of the reports in the local store.

```
$> make list
[+] reports:
[+]   20101103:140601:
[+]     xen :: ibm-pc :: ia32/educational :: k2 :: 2010/11/03 14:06:01
```

The *display* command gives the user the possibility to dump a locally stored report in a very detailed way.

```
$> make display-20101103:140601
[+] report:
[+]   meta:
[+]     platform:               ibm-pc
[+]     date:                   2010/11/03 14:06:01
[+]     architecture:           ia32/educational
[+]     duration:               63.499
[+]     suite:                  k2
[+]     identifier:             20101103:140601
[+]     environments:
[+]       stress:               xen
[+]       construct:            xen
```

```
[+]   data:
[+]     segment:                                                    [1/1]
[+]       permissions/01:
[+]         status: True
[+]         description: This test creates a task and address space before reserving a seg-
ment and changing its permissions.
[+]         duration: 0.010
[+]         output:
[+]     id:                                                         [3/3]
[+]       simple:
[+]         status: True
[+]         description: This test reserves a single identifier.
[+]         duration: 0.004
[+]         output:
[+]       clone:
[+]         status: True
[+]         description: This test reserves, clones and releases identifiers.
[+]         duration: 0.005
[+]         output:
[+]       multiple:
[+]         status: True
[+]         description: This test reserves thousands of identifiers, checking that no col-
lisions occured.
[+]         duration: 0.040
[+]         output:
$>
```

The *submit* command sends the user's snapshot to the server so as to be evaluated for the given stage.

```
$> make submit-k1
[+] configuration:
[+]   server:               https://test.opaak.org:8421
[+]   capability:           /data/mycure/repositories/kaneton/environment/profile/user/julien.quintard/juli
[+]   platform:             ibm-pc
[+]   architecture:         ia32/educational

[+] the snapshot has been submitted successfully
$>
```

Finally, the *retest* command provides contributors the possibility to re-launch the test suite of the given identified test. This command is especially useful to re-test a snapshot should an unexpected error occur on the test server.

Indeed since test requests are limited for students, it would be unfair for the student to be forced to sacrifice a test slot because something went wrong on the server-side. By requesting a contributor, the student's snapshot can be re-tested. Once the test complete, an email is sent to the student along with the attached report.

```
$> make retest-20101103:140601
[+] configuration:
[+]   server:               https://test.opaak.org:8421
[+]   capability:           /data/mycure/repositories/kaneton/environment/profile/user/julien.quintard/juli
[+]   platform:             ibm-pc
[+]   architecture:         ia32/educational

[+] the snapshot has been re-tested successfully
$>
```

Robot

The *robot* test tool enables contributors to test the kaneton research implementation on a regular basis; hence control the status of the development.

The robot basically retrieves the kaneton implementation by checking out the *Subversion* repository. Then, several test suites are triggered through the test client. Once the reports have been received, a message is built summarizing the results. This message is then sent to the kaneton contributors mailing-list.

The deployment of the *robot* is quite straigthforward. First, the `test/robot/` directory must be copied to the server. Note that the *robot.py* script depends upon the *ktp* package which must therefore be copied as well.

Then, the *SSH* configuration file *config* must be placed in the `$HOME/.ssh/` directory. Besides, this file should be edited in order to properly reference the *SSH* keys since the default configuration assumes the kaneton test directory to be located at `/kaneton/`.

Finally, the *robot.cron* crontab file must be setup through the following command in order to trigger the robot every night:

```
$> crontab robot.cron
```

Once again, the administrator should make sure to edit this file should the robot files not be located in the default location *i.e.* `/kaneton/`.

5.1.8 Prototypes

The `tool/` directory contains a interesting tool developed by the kaneton community for generating *C* prototypes automatically.

This tool named *mkp* must not be used directly as the prototypes automatic generation can be triggered from the *Make* files via the *prototypes* rule. This rule should be called everytime a developer wants the prototypes to be correct, for instance, before starting a compiling process.

This tool was introduced to avoid spending time writing prototypes. Since, this utility generates the prototypes automatically, developers should no longer write prototypes manually.

This script takes a list of *C* header files as arguments. These header files are supposed to contain a *prototypes* section as illustrated below:

```
/*
 * ------- prototypes -----------------------------------
 *
 *      ../../core/set/set.c
 *      ../../core/set/set-array.c
 *      ../../core/set/set-ll.c
 *      ../../core/set/set-bpt.c
 *      ../../core/set/set-pipe.c
 *      ../../core/set/set-stack.c
 */

[...]

/*
 * eop
 */
```

The *mkp* tool then retrieve the list of the *C* source files listed in the *prototypes* section and remove the text between the *prototypes* opening section and the `eop` tag.

Then, the script goes through each of the listed source files and tries to detect *C* function declarations. For each function declaration found, the tool generates a prototype in the corresponding header file.

Below is an example of a header file with such a *prototypes* section and generated prototypes.

```
/*
 * ------- prototypes ----------------------------------
 *
 *      ../../core/set/set.c
 *      ../../core/set/set-array.c
 *      ../../core/set/set-ll.c
 *      ../../core/set/set-bpt.c
 *      ../../core/set/set-pipe.c
 *      ../../core/set/set-stack.c
 */

/*
 * ../../core/set/set.c
 */

t_error                 set_dump(void);

t_error                 set_size(i_set                          setid,
                                 t_setsz*                       size);

t_error                 set_new(o_set*                          o);

t_error                 set_destroy(i_set                       setid);

t_error                 set_descriptor(i_set                    setid,
                                       o_set**                  o);

t_error                 set_get(i_set                           setid,
                                t_id                            id,
                                void**                          o);

t_error                 set_init(void);

t_error                 set_clean(void);


/*
 * ../../core/set/set-array.c
 */

t_error                 set_type_array(i_set                    setid);

t_error                 set_show_array(i_set                    setid);

t_error                 set_head_array(i_set                    setid,
                                       t_iterator*              iterator);

[...]

/*
 * eop
 */
```

5.1.9 Control Panel

The kaneton environment allows the developer to trigger every action from the *Make* file of the project's root directory.

These actions are listed below:

`make initialize`

This action initializes the kaneton development environment by invoking the `init.py` script of the `environment/` directory.

`make clean`

This action cleans the kaneton development environment.

`make main`

This action triggers the default rule which aims at compiling every piece the final system needs to be set up on a bootable device.

Example:

```
$ make main
```

Example:

```
$ make
```

`make clear`

This action removes every compiled files.

`make headers`

This action generates *Make* files' *C* header files dependencies.

`make prototypes`

This action generates C prototypes.

`make test`

This action runs the test suite in order to validate the kaneton microkernel behaviour.

`make cheat`

This action launches the cheat tests on students kaneton implementations.

Example:

```
$ make cheat
```

Example:

```
$ make cheat-EPITA::2006::k3
```

`make build`

This action builds the boot device.

`make install`

This action installs the kaneton microkernel with its dependencies: configuration files, bootloader *etc.* on the boot device.

`make export`

This action builds a kaneton distribution package.

Example:

```
$ make export
```

Example:

```
$ make export-k3,5
```

Example:

```
$ make export-back
```

`make view`

This action builds and displays a kaneton document.

Example:

```
$ make view
```

Example:

```
$ make view-devel
```

Example:

```
$ make view-book::kaneton
```

`make record`

This action records a real-time session.

Example:

```
$ make record
```

Example:

```
$ make record-basic::test.ts
```

`make play`

This action plays a recorded session.

Example:

```
$ make play
```

Example:

```
$ make play-basic::prototypes.ts
```

Example:

```
$ make play-prototy
```

`make info`

This action displays general information about kaneton.

5.2 External

The kaneton contributors use several other tools for the communication, the development *etc.* These tools are described in the following sections.

5.2.1 Mailing-List

Because people do not want to use several communication tools: email, newsgroup, forum *etc.* and because everybody has an email address, kaneton people communicate through a mailing-list.

The kaneton project relies on two distinct communication groups:

- `contributors@kaneton.opaak.org` is dedicated to the communication between the people involved in both the development and teaching of the kaneton microkernel project.

  This group is therefore private.

- `students@kaneton.opaak.org` can be used in an absolutely free-way by students for communicating about their kaneton educational implementation.

  Anybody can join this group.

Needless to say, community rules discussed in *Chapter 3* must be followed when communicating on the kaneton mailing-lists. Every contributor is welcome to give her point of view, to ask questions *etc.* but this must be done with politness, respect and humility.

Everyone communicating through the mailing-list must read the *Netiquette* which describes the rules inherent to the communication on the Internet. Especially, people should take care of writing messages in respect of the 80 columns; and should always cut off useless parts in previous messages when responding.

It is likely that a real-time communication tool will be very useful in the future, an *IRC* channel for instance. However, communicating on these extra media will not be mandatory unless kaneton people decide so.

kaneton people are asked to use the mailing-list communication medium in a perfect way as it is the unique intra-communication channel. In addition then, contributors must read their emails on a regular-basis as some people rely on the decision of others.

The *students* mailing-list must be used carefully as well. As an example, people should never paste pieces of source code or ask questions implying an answer with the solution. Even if it is a free group, people abusing of this communication channel could be easily banned.

People must always respond in the appropriate discussion. If, in a discussion, a different subject is discussed, then, one of the contributor must create a new discussion in order facilitate the communication.

Finally, the discussion subjects must be tagged like the following examples:

```
[ia32/optimised] mapping issues

[segment] segment_clone() :: bug?

[research] new paper about OS design
```

There is no list of official tags. The users are simply asked to make their discussion subjects as clear as possible.

5.2.2 Repository

The repository contains everything related to the kaneton microkernel project, in other words, the kaneton source tree described in *Chapter 2*. Indeed, the repository contains the whole history of the kaneton project including the documentation, the source code but also the students tarballs over the years.

The actual repository is based on the *Subversion* software which provides far more advanced features than its historical rival *CVS*.

The repository is actually hosted on the `kaneton.opaak.org` server which also contains the web server and everything else related to the management of the kaneton microkernel project.

The repository is accessed in a secure way through a *SSH* channel. Indeed, the kaneton *Subversion* repository can be accessed at the following address: `svn+ssh://subversion@repositories.opaak.org/kaneton`.

Note that the security is achieved by the use of *SSH* keys. Therefore, any new contributor should get in touch with an administrator of the kaneton server in order to obtain an access. Also note that, a test period could be set up for a new contributor to get the trust of the kaneton community. For more information, please refer to *Chapter 3*.

A contributor willing to create a *SSH* key shoud simply use this *Unix* command:

```
$> ssh-keygen -t dsa
```

For more information about how to use the repository, please refer to the official *Subversion* documentation. The same goes for the *SSH* tools suite.

The example below illustrates the checkout of the kaneton repository.

```
$> svn checkout svn+ssh://subversion@repositories.opaak.org/kaneton
```

The contributors getting access to the kaneton repositories must behave properly according to the obvious cooperative development rules. As an example, a kaneton developer must not perform any commit before making sure the kaneton microkernel compiles and passes all the tests.

The repository organisation is crucial. Therefore, nothing should be added, removed or renamed without the permission of the developers in charge of the repository.

Finally, any commit must come with a log describing the modifications implied by the commit. These logs must conform to the following syntax.

```
[kaneton/core/segment/]
  o the bug about the permissions was corrected in segment_clone().
  o an algorithm based on a b-tree was added.

[environment/profile/user/julien.quintard/]
  o some personal configurations were modified.
```

Following this syntax is very important as an email is sent to the *kaneton-developers* mailing-list every time a commit is performed. Therefore, the contributors reading the mailing-list are aware of every modification in the kaneton source code. This feature can also be used to review the modifications done by a new contributor in order to help him doing things in a better way.

Note that there must not be any file with the executable flag permission enabled. Moreover, scripts files must not contain any *shebang*. Indeed, the kaneton development environment knows which interpreter to use for every type of file. It is therefore a non-sense to introduce a hard-coded path to an interpreter.

Tarball file names must be extended with `.tar` while *bzip2* compressed tarballs must be extended with `.tar.bz2`.

5.2.3 Wiki

A Wiki is used both for external and internal communication. The software used is called *TWiki* and provides a pretty simple syntax with many additional plugins to customize the website. This solution was used for historical reasons but also because the *TWiki* rendering can be customized through templates in order to get a final visual close to classical websites. Thus, the kaneton website looks like an ordinary website but powered by a Wiki engine.

The Wiki is hosted at `http://kaneton.opaak.org` and contains two webs: an extranet and an intranet. The main web, accessed through the address above is the external website. This website contains news, papers, documentation and general information on the kaneton project. The other web is used more as an intranet or wiki more than as a website.

Finally, http://wiki.opaak.org is intended to students and contains documents, links *etc.* about low-level programming, kernel development *etc.* as well as information about courses related to the *opaak* projects.

Everybody involved in the kaneton project must contribute to the kaneton website as well as to the kaneton intranet and wiki. Indeed, the external communication is fundamental, even in an open source project and the kaneton website is the only public communication medium.

New contributors are then asked to register onto the kaneton *TWiki* at http://kaneton.opaak.org. Once done, the contributor should inform the person in charge of the kaneton website so that the contributor's account is activated. As a result, the contributor will be able to modify pages of the website and intranets.

5.2.4 Project Management

It is very important to really well understand that kaneton is a community-driven project. Hence, people cannot expect to always do what they would like to as many things need to be done, fixed or whatever.

Additionally, as the project grows, it needs to be more carefully managed so that bugs are fixed, boring tasks are done *etc.*

A project management tool is provided in order to better manage tasks according to their priorities. *Trac* was picked for this purpose as it was probably, at that time, the best choice.

The project management tool is running on the kaneton.opaak.org server and authentication is based on the *Wiki*.

Always remember that, although *Trac* provides a *Wiki* as well, the kaneton.opaak.org *Developers Intranet* must be used instead.

Chapter 6

Languages

This chapter focuses on the languages in the context of the kaneton microkernel project. Indeed, many different languages are used for the microkernel itself, but also for the scripts and documentations around the project.

6.1 Make

The coding style related to *Make* files is very simple.

6.1.1 Naming

First, *Make* files must be named `Makefile`, with the first letter in upper case letter.

Rules must be named with a single word in lower-case letters while variables must be named in upper-case letters.

6.1.2 Environment

Every *Make* file must rely on the development environment as it provides a way for performing operations in a very simple and portable way.

Therefore, each *Make* file must start by including the configured *Make* environment file located in the `environment/` directory under the name: `env.mk`.

Note that the configured environment *Make* file provides a hidden rule named `_`. This rule name is therefore reserved. This rule is triggered every time a *Make* file is called without specifying a rule name. This rule aims at verifying the configured environment is up-to-date. If not, the rule regenerates it. Finally, this rule calls the `main` rule to perform the main work of the *Make* file.

Therefore, every *Make* file has to provide a `main` rule.

6.1.3 Layout

Make files must be organised as sections.

- **header**: this section contains the file header which provides information on the file edition: creation, last update *etc.*
- **dependencies**: this section contains the `include` directives.

```
#
# ------- dependencies --------------------------------
#

include                 ../../environment/env.mk

[...]

#
# ------- dependencies --------------------------------
#

-include                ./$(_DEPENDENCY_MK_)
```

- **directives**: this section contains *Make* directives like `.PHONY`, `.SUFFIXES` *etc.*

```
#
# ------- directives ---------------------------------
#

.PHONY:         clear prototypes headers dependencies
```

- **variables**: this section contains the variable definitions.

```
#
# ------- variables ----------------------------------
#

SUBDIRS                 :=              arch                    \
                                        kernel                  \
                                        as                      \
                                        conf                    \
                                        id                      \
                                        region                  \
                                        segment                 \
                                        debug                   \
                                        map

CORE_INCLUDE            :=              $(_CORE_INCLUDE_DIR_)/core.h

CORE_C                  :=              core.c

CORE_O                  :=              $(CORE_C:.c=.o)
```

- **rules**: this section contains the *Make* rules.

```
#
# ------- rules --------------------------------------
#

$(LIBSTRING_LO):        $(LIBSTRING_O)
        $(call env_remove,$(LIBSTRING_LO),)

        $(call env_archive,$(LIBSTRING_LO),$(LIBSTRING_O),)

clear:
        $(call env_remove,$(LIBSTRING_O),)

        $(call env_remove,$(LIBSTRING_LO),)

        $(call env_purge,)

prototypes:
        $(call env_prototypes,$(LIBSTRING_INCLUDE),)

headers:
        $(call env_remove,$(_DEPENDENCY_MK_),)

        $(call env_headers,$(LIBSTRING_C),)
```

6.1.4 Style

Recall that the kaneton development environment provides a *Make* interface allowing *Make* files to perform operations in a portable way. Thus, *Make* files must not contain any operations specific to any operating system.

As such, every operation should be a call to one of the *Make* functions provided by the kaneton development environment interface.

Any contributor writing a *Make* file should take care of properly cleaning the directory from temporary files. Indeed, every *Make* file provides a `clear` rule which is intended to remove these temporary files. Therefore, every developer should take care of that and verify every file was cleaned before committing.

The kaneton project provides a tool for generating *C* prototypes through a *Make* rule called `prototypes`. Therefore, the *Make* file must carefully use this rule to trigger the generation of prototypes in the right header files. For more information about the prototypes generation, please refer to *Section 5.1.8*.

One of the other important rules is called `headers` and generated a file containing all the header file dependencies related to a set of *C* source files. This generated file must be included in the *Make* so that it can use it. A *Make* file inclusion is present at the end of many *Make* files for this purpose:

```
#
# ------- dependencies ---------------------------------
#

-include                ./$(_DEPENDENCY_MK_)
```

Additionally, *Make* files must resolve dependencies before performing any other task. Therefore, the rule name `dependencies` is reserved for this purpose by launching every *Make* file in charge of one of the dependency.

Finally, recall that a `main` rule must be provided by every *Make* file since this rule is called everytime the *Make* file is launch without rule name argument.

6.2 Python

The *Python* language is the other language for which the kaneton development environment provides an interface for performing task in a portable way.

Historically, kaneton scripts were written in *Shell* but scripts are pretty hard to write, especially when dealing with file modification. Even if *sed*, *awk etc.* tools are very powerful, this is not a good solution for many reasons. Therefore, the scripts were re-written in *Python* whilst a *Python* development environment interface was written.

6.2.1 Naming

File names but also function names must be expressed in lower-case letters and a dash must be used for composite names.

Comments must also be written in lower-case letters.

Global variable names, which are often used in *Python*, must be prefixed by `g_`.

As usual, names must be expressed in a single word as long as this is possible. Moreover, the shortcut prefixes and suffixes recommended for the *C* language also stand for *Python*. For more information, take a look at the *Section 6.4*.

6.2.2 Environment

Every *Python* file must rely on the development environment as it provides a way for performing operations in a very simple and portable way.

Therefore, each *Python* must start by including the configured *Python* environment module located in the `environment/` directory under the name: `env.py`.

Also, since operations are provided by the development environment, note that *Python* scripts must not perform operations relying on the `sys` and/or `os` modules. Others modules like `re`, `yaml` *etc.* are allowed. Moreover, the `sys` module is sometimes used - badly. If you do not know whether the usage of a module is allowed, please ask your supervisor or directly the kaneton community through the mailing-list for instance.

6.2.3 Layout

The *Python* files must be divided into sections. Below are listed the most common sections developers should use.

- **header**: this section contains the well-known informations about the file: dates, author, license *etc.*
- **information**: this section describes what the *Pyhton* do or provide if it is a package.
- **imports**: this section contains the module imports.

```
#
# ------- imports ----------------------------------
#

import env

import sys
import re
```

- **globals**: this section contains the global variable declarations.

```
#
# ------- globals ----------------------------------
#

g_directories = (‘book’,
                 ‘curriculum’,
                 ‘exam’,
                 ‘feedback’,
                 ‘internship’,
                 ‘lecture’,
                 ‘paper’)
g_store = []
g_document = None
g_path = None
```

- **functions**: this section contains the actual *Python* functions.
- **entry point**: this section contains the real entry point.

  This section was introduced in order to localise the script's entry point more easily. Note that this section is generally located at the end of the script.

```
#
# ------- entry point ---------------------------------
#

if __name__ == '__main__':
  main()
```

6.2.4 Style

Every function must be preceded by a comment. This comment plays the role of visual separator but it also describes what the function does.

This comment first contains the name of the function with two parentheses. Then follows an empty comment line and the function description.

```
#
# usage()
#
# this function prints the documents names.
#
def                     usage():
  location = None
  store = None

  env.display(env.HEADER_ERROR, 'usage: view.py [document]',
              env.OPTION_NONE)

  env.display(env.HEADER_NONE, '', env.OPTION_NONE)

  env.display(env.HEADER_ERROR, 'documents:', env.OPTION_NONE)

  for store in g_store:
    for location in store:
      env.display(env.HEADER_ERROR, '  ' + location, env.OPTION_NONE)
```

The identation used in *Python* script must be fixed at 2 whitespaces. This is a common kaneton rule which allows files to be readable while respecting the limitation about the 80 characters in width.

Finally, note, that in *Python*, variables do not need to be declared before being actually used. However, kaneton people believe this lead to unreadable code as well as to common hard-to-find bugs. For this reason, every variable must be declared at the beginning of the function. Finally, these variable should also be initialized.

6.3 Assembly

The *Assembly* language is used in kaneton to perform tasks which cannot be done in *C*. Therefore, the amount of *Assembly* code is relatively small compared to the *C* one.

6.3.1 Inline Assembly

First of all, inline *Assembly* is recommended as it avoids multiple files and calls to *Assembly* routines.

Inline *Assembly* code must be composed of lines with single instructions. Lines must be ended by a newline character \n and aligned to make the code readable even when nested in the body of a macro or macro function.

```
#define EXCEPTION_PREHANDLER_CODE(nr)                                   \
  asm(‘.globl prehandler_exception’ #nr ‘           \n’           \
      ‘prehandler_exception’ #nr ‘:                 \n’           \
      SAVE_CONTEXT()                                                \
      ‘iret                                         ’)
```

Note that rules implied by the *C* language must be respected when dealing with inline *Assembly* as these pieces of code are integrated in *C* functions.

6.3.2 Naming

Bunches of coherent routines must be preceded by a comment, starting with the name of the routine in upper case letters, and followed by a description of the routines, in lower-case letters.

6.3.3 Layout

An *Assembly* file must be composed of sections:

- **header**: this section contains the file header which provides information on the file edition: creation, last update *etc.*
- **information**: this section describes what the file is intended to do or provide.
- **routines**: this section contains the *Assembly* routines.
- **data**: this section contains the data definitions.

Below is an example:

```
;
; ------- information ---------------------------------
;
; this is a example, no additional information is required.
;

;
; ------- routines ------------------------------------
;

;
; PRINT STRING
;
; these routines print a string.
;

print_string:
        mov ah, 0x0e            ; the function to call with int 0x10
        mov bx, 0x07            ; the console attributes

print_string_loop:
        lodsb                   ; load a character from esi into al
        cmp al, 0               ; is the string finished?
```

```
        je print_string_done    ; if true jump to the end

        int 0x10                ; ask the BIOS to perform the task

        jmp print_string        ; loop for the next character

print_string_done:
        ret                     ; return 'cause the string is now displayed

;
; MAIN
;
; the main routine.
;

main:
        mov [bootdrive], dl     ; save the bootdrive identifier
        xor ax, ax              ; initialize ax
        mov ds, ax              ; initialize ds

        cli
        mov ss, ax              ; initialize ss
        mov sp, 0xffff          ; initialize the stack pointer
        sti

        mov si, newline
        call print_string

        mov si, rmode_message
        call print_string

        call floppy_read        ; read the ELF from the floppy

        call pmode_enable       ; enable the protected mode

                                ; once the protected mode is enabled
                                ; the function pmode_main will be launched

;
; ------- data ---------------------------------------
;

newline         db      10, 0

rmode_message   db      '[+] real mode', 13, 10, 0

bootdrive       db      0
```

6.4 C

This section is intended to detail the kaneton *C* coding style. The specifications in this document are to be well-known by every kaneton developer.

The kaneton coding style was introduced in order to uniformize the *C* coding styles of kaneton project contributors. Indeed, as explained in this section, the kaneton coding style is very different from the coding styles of other open source projects especially the *GNU* style.

The key aspect of the coding style is the clarity through the coherency. Indeed, every choice was made for the good of the project so that everyone can very easily read the source code of the kaneton microkernel.

6.4.1 Naming

Names in kaneton must comply to the following rules.

General

Entities generally belong to a higher component: manager, module, package *etc.* Since the *C* language does not provide any namespace feature, the programmer must rigorously name entities by prefixing them with the name of the higher components.

Composite names must be separated by underscores: `_`.

Moreover, entities including variables, functions, macros, types *etc.* must have explicit and/or mnemonic names.

```
#define IA32_OPTION_READ              1

#define SET_TYPE_ARRAY                0x02

#define MIPS64_THREAD_STACKSZ         8192
```

Obviously, names can be abbreviated, but only when it allows shorter code without a loss of meaning. The following rules must be used in this way:

- The suffix `sz` must be used to represent a size:

```
#define PAGESZ                    4096

int                               modsz;
```

- The `s` suffix must be used for representing mutiple entities:

```
int                               threads;
t_uint64                          sets;
```

- The `n` prefix must be used for variables representing a number:

```
int                               ntasks;
```

More generally, entities must be named with a *unique* word, excluding the namespace prefix. Short less explicit names are always prefered on very long explicit names which are then difficult to name and use.

The following example illustrates a wrong usage of names in kaneton:

```
t_errval              segment_dump_set(i_set              segment_set,
                                       t_uint32           number_of_segmens);
```

In this example, many entities are badly named and should be replaced by something similar to the following:

```
t_error                 segment_dump(i_set              segments,
                                     t_uint32           nsegments);
```

Indeed, types are used as pre-names. Therefore, the entity name should not overlap the type name. For example, the following example is incorrect:

```
t_vaddr                 video_addr;

i_event                 eventid;
```

Developers should prefer simpler names:

```
t_vaddr                 video;

i_event                 id;
```

These rules must be respected, especially in the case of functions belonging to an interface. Indeed, since interfaces are presented to the user, they must be easy to use and self-described. Therefore, functions as well as arguments must be very clearly named, using a single word.

Finally, names must be expressed in English, without spelling mistakes.

Capitalization

Entities including variables, functions, types, structures, enumerations, unions *etc.* must be expressed using lower-case letters, digits and underscores only. More precisely, entity names must be matched by the following regular expression:

```
[a-z][a-z0-9_]*
```

This is the common rule. However, some exceptions exist.

Macro names must be entirely capitalized except for macros which create an abstraction as it is the case for the *set manager* interface. Macro arguments, must be prefixed and suffixed by an underscore so that naming collision is avoided.

Note that the kaneton coding style names function arguments as **arguments** and macro arguments as **parameters** as these latter ones are statically computed.

```
#define SCHEDULER_STATE_RUN           0

#define SEGMENT_BPT_NODESZ            4096

#define set_reserve(_type_, _args_...)                          \
  set_reserve_##_type_(_args_)
```

Comments must be written in lower-case letters. Indeed, no capitalization must be used, at all.

```
/*
 * this function shows a segment.
 *
 * steps:
 *
```

```
 * 1) get the segment object.
 * 2) compute the type string.
 * 3) compute the perms string.
 * 4) display the entry.
 * 5) call machine dependent code.
 */

t_error                 segment_show(i_segment                  id)
{
  [...]
}
```

Types

Type names are classified according to the group they belong to.

Firstly, structures, unions and enumerations must not be directly used. Instead, a type must be defined. In order to indicate the entity class a type represents, prefixes are used.

Note that the core elements do not need prefixing since considered as the most fondamental elements of the kaneton microkernel. However, several element markers are used to distinguish the different kaneton types:

- Managers are prefixed by `m_`.
- Identifiers are prefixed by `i_`.
- Object types are prefixed by `o_`;
- Dispatch tables are prefixed by `d_`;

Then, the other components must rely on namespace prefixes in order to avoid conflicts:

- The glue component's types are prefixed by `g_`;
- The platform-dependent component's types are prefixed by `p_`;
- The architecture-dependent component's types are prefixed by `a_`;
- The module-dependent component's types are prefixed by `m_`;
- The library-dependent component's types are prefixed by `l_`;

Finally, should none of the above definitions match the defined type, the general purpose types listed below can be used:

- Function pointers are prefixed by `f_`;
- Structures are prefixed by `s_`;
- Enumerations are prefixed by `e_`;
- Unions are prefixed by `u_`;
- General purpose aliased-types are prefixed by `t_`.

Note that prefixes can be combined in several ways. For instance, the prefix `ao_` represents an architecture-dependent object while the `lt_` represents a library type.

The example below illustrates this rule:

```
typedef void                    (*mf_console_character)(char);
typedef void                    (*mf_console_attribute)(t_uint8);

typedef struct
{
  mf_console_character          character;
  mf_console_attribute          attribute;
}                               mm_console;
```

The use of *C* standard types like `int`, `long long` is prohibited since there exist kaneton-specific types `t_uint32`, `t_uint64` which provide a size guarantee.

6.4.2 Layout

The global layout of files and sections of code pertaining to the *C* preprocessor, including file inclusion and inclusion protection, must comply to specifications detailed in the following.

Note that an *Emacs* configuration file is provided in the developers' private *Wiki*. This file contains bindkeys for generating many of the syntaxes explained in this section. If some contributors are using a different text editor, then it is their responsibility to follow the rules by developing a plugin for their text editor, for instance.

File

C source and header files are composed of sections. The example below illustrates these sections:

```
/*
 * ----- dependencies ---------------------------------
 */

#include <core/id.h>
#include <core/types.h>

/*
 * ----- macros ---------------------------------------
 */

/*
 * types
 */

#define SET_TYPE_ARRAY          0x01
#define SET_TYPE_BPT            0x02
#define SET_TYPE_LL             0x03
#define SET_TYPE_PIPE           0x04
#define SET_TYPE_STACK          0x05

/*
 * iterator's state
 */

#define ITERATOR_STATE_USED     0x01
#define ITERATOR_STATE_UNUSED   0x02
```

Next are listed some of the most important sections:

- **header**: this section contains the file header which provides information on the file edition: creation, last update *etc.*

  Note that the first and last authors are also specified in this header. Remember that kaneton is a community driven project and therefore these names do not represent the main file's authors.

  This section must be present in every source and header file as it contains the file creation and last updates.

- **information**: this section contains a general description of what this file provides.

  This section must be present in every source file.

```
/*
 * ------- information ------------------------------
 *
 * the address space manager manages address spaces.
 *
 * an address space describes process' useable memory. each address space
 * is composed of two sets.
 *
 * the first describes the segments held by this address space, in other
 * words the physical memory.
 *
 * the latter describes the regions, the virtual areas which reference
 * some segments.
 *
 * a task can give its address space to another with as_give().
 */
```

- **dependencies**: this section contains inclusions of dependency files.

  This is the common section which contains the well-known `#include` preprocessor directives.

- **includes**: this section contains inclusion statements of additional files.

  This section must not be misused since there also exist a *dependencies* section.

```
/*
 * ------- dependencies -----------------------------
 */

#include <core/id.h>
#include <core/types.h>

/*
 * ------- includes ---------------------------------
 */

#include <core/set-array.h>
#include <core/set-bpt.h>
#include <core/set-ll.h>
```

- **macros**: this section contains macro definitions.

  This section must not contain macro functions as there is a section dedicated to these.

- **macro functions**: this section contains the macro function definitions which are dissociated from the basic macro definitions.

```
/*
 * ------- macros -----------------------------------
 */

/*
 * options
 */

#define SET_OPT_NONE            0x00
#define SET_OPT_FORWARD         0x01
#define SET_OPT_BACKWARD        0x02
#define SET_OPT_CONTAINER       0x04
#define SET_OPT_ALLOC           0x08
#define SET_OPT_FREE            0x10
#define SET_OPT_SORT            0x20
#define SET_OPT_ORGANISE        0x40

/*
 * ------- macro functions ----------------------------
 */

#define set_type(_type_, _id_)                                  \
  set_type_##_type_(_id_)

#define set_reserve(_type_, _args_...)                          \
  set_reserve_##_type_(_args_)
```

- **types**: this section contains de type definitions.

- **prototypes**: this section contains the prototype definititions.

  This section is specific as the kaneton microkernel project uses a tool for generating prototypes from *C* source file. For more information, please refer to *Section 5.1.8*.

```
/*
 * ------- prototypes --------------------------------
 *
 *      ../../core/set/set.c
 *      ../../core/set/set-array.c
 *      ../../core/set/set-ll.c
 *      ../../core/set/set-bpt.c
 *      ../../core/set/set-pipe.c
 *      ../../core/set/set-stack.c
 */

/*
 * ../../core/set/set.c
 */

t_error                 set_dump(void);

t_error                 set_size(i_set                          id,
                                 t_setsz*                       size);
```

- **externs**: this section contains external declarations.

```
/*
 * ------- extern -----------------------------------
 */
```

```
/*
 * the init variable, filled by the bootloader, containing in this case
 * the list of segments to mark used.
 */

extern t_init*          init;
```

- **globals**: this section contains global variable declarations.

```
/*
 * ------- globals ---------------------------------
 */

/*
 * the segment manager structure.
 */

m_segment*              segment;
```

- **functions**: this section contains function definitions.

More specifically, *C* header files are likely to contain the sections: *header*, *dependencies*, *macros*, *types*, *includes*, *macro functions*, *prototypes etc.* while the *C* source files should contain the sections: *header*, *information*, *includes*, *externs*, *globals*, *functions etc.*

The *header* section is basically the file header which was described in *Chapter 4*.

Preprocessor

The preprocessor directives must appear on the first column with no identation:

```
#ifndef CORE_SET_H
#define CORE_SET_H              1

/*
 * debug
 */

#if (DEBUG & DEBUG_SET) && defined(SET_DEBUG_TRAP)

#define set_debug(_func_, _id_, _args_...)                              \
  fprintf(stderr, '[setd] trap: %s(%qu, %s)\n',                         \
          #_func_,                                                      \
          _id_,                                                         \
          #_args_);

#else

#define set_debug(_func_, _id_, _args...)

#endif

#endif
```

All header files must be protected against multiple inclusions. The guard macro must be named according to the location of the header file with the suffix `_H` in order to avoid guard macro name collision.

```
#ifndef ARCHITECTURE_IA32_SEGMENT_H
#define ARCHITECTURE_IA32_SEGMENT_H           1

[...]

#endif
```

Spanning macros over multiple lines is encouraged for the sake of clarity. In such cases, escaped line breaks `\-newline` must appear on the same column. For this purpose, tabulations must be used.

Moreover, the body of macro functions must start on a new line, idented by 2 whitespaces.

```
/*
 * traps
 */

#define set_trap(_func_, _id_, _args_...)                               \
  (                                                                     \
    {                                                                   \
      t_error           _r_ = ERROR_UNKNOWN;                          \
      o_set*            _set_;                                        \
                                                                      \
      set_debug(_func_, _id_, _args_);                                \
                                                                      \
      if (set_descriptor((_id_), &_set_) == ERROR_NONE)               \
        {                                                             \
          switch (_set_->type)                                        \
            {                                                         \
              case SET_TYPE_ARRAY:                                    \
                _r_ = _func_##_array((_id_), ##_args_);               \
                break;                                                \
              case SET_TYPE_BPT:                                      \
                _r_ = _func_##_bpt((_id_), ##_args_);                 \
                break;                                                \
              case SET_TYPE_LL:                                       \
                _r_ = _func_##_ll((_id_), ##_args_);                  \
                break;                                                \
              case SET_TYPE_PIPE:                                     \
                _r_ = _func_##_pipe((_id_), ##_args_);                \
                break;                                                \
              case SET_TYPE_STACK:                                    \
                _r_ = _func_##_stack((_id_), ##_args_);               \
                break;                                                \
            }                                                         \
        }                                                             \
      _r_;                                                            \
    }                                                                 \
  )
```

Functions

Every function must be preceded by a comment which completely describes the actions the function performs. Moreover, some function parts need additional descriptions. In order not to overload the function body with heavy comments, a kaneton-specific function organisation was introduced.

Indeed, functions body are composed of **steps**, each step representing a bunch of coherent actions. These actions are not described in the body of the function, but rather in the comment preceding the function. Thus, the body is clearly decomposed into steps for the sake of clarity while expressiveness is guaranted by heavy documentation contained in the function comment.

Therefore, function definitions must comply the organisation illustrated below:

```
/*
 * this function adds a set descriptor to the set container.
 *
 * steps:
 *
 * 1) if the descriptor to add is the set which will contain the set objects,
 *    the container, just put it as the set container.
 * 2) otherwise, add this object in the set container.
 */

t_error                 set_new(o_set*                          o)
{
  SET_ENTER(set);

  /*
   * 1)
   */

  if (o->setid == set->sets)
    {
      if ((set->container = malloc(sizeof(o_set))) == NULL)
        SET_LEAVE(set, ERROR_UNKNOWN);

      memcpy(set->container, o, sizeof(o_set));

      SET_LEAVE(set, ERROR_NONE);
    }

  /*
   * 2)
   */

  if (set_add(set->sets, o) != ERROR_NONE)
    {
      cons_msg('!', 'set: unable to add this set descriptor '
               'to the set container\n');

      SET_LEAVE(set, ERROR_UNKNOWN);
    }

  SET_LEAVE(set, ERROR_NONE);
}
```

Note that a single blank line must be put for separating pieces of code as it leads to a more readable source code. Moreover, a blank line must be put before and after each step comment in the function's body.

Note that comments must comply to the template exposed above. The following are examples of bad comments:

```
/*
** bad comment
*/

/* Bad Comment
 */

/* bad comment */

// bad comment
```

6.4.3 Style

The following sections specify various aspects of what constitutes good programming behaviour at the language level. They cover various aspects of the *C* constructs.

Blocks

All braces must be on their own line. This rule implies a programming style very different from the *BSD* or *GNU* coding styles which put the open brace at the end of the line.

```
if (option & OPTION_OPTIMISED)
  {
    [...]
  }
```

In addition, closing braces must appear on the same column as the corresponding opening brace.

The text between two braces must be indented by a fixed, homogeneous amount of whitespaces. This amount is fixed to 2 whitespaces. Note that the *Emacs* default identation comply to this rule.

Moreover, the braces must also be indented by the amount of 2 spaces from the previous line. However, some exceptions exist, especially with nested block declarations.

In *C* functions, the declaration part must be separated from statements with a single blank line. Note that when there are no declarations, there must not be any blank line separator.

Blocks are generally composed of bunches of statements and expressions. Every developer is welcomed to put a single blank line in order to clearly separate bunches of pieces of code.

```
t_error                 set_get(i_set                           setid,
                                t_id                            id,
                                void**                          o)
{
  t_iterator            iterator;

  SET_ENTER(set);

  if (set_locate(setid, id, &iterator) != ERROR_NONE)
    SET_LEAVE(set, ERROR_UNKNOWN);

  if (set_object(setid, iterator, o) != ERROR_NONE)
    SET_LEAVE(set, ERROR_UNKNOWN);

  SET_LEAVE(set, ERROR_NONE);
}
```

Alignment

Declaration identifiers must be aligned with the function name, using tabulations only. Moreover, the declarations must be ordered according to the length of the identifier, starting with the longer, as shown below.

```
t_error                 segment_dump(void)
{
```

```
  t_state               state;
  t_setsz               size;
  t_iterator            i;

  [...]
}
```

In *C*, pointerness is not part of the type. However, in the kaneton coding style, the * pointer symbol in declarations must appear next to the type.

Function argument lists must be broken between each argument, after the comma. In addition, the argument identifiers must be properly aligned together, with tabulations.

```
t_error                 as_give(i_task                  tskid,
                                i_as                    asid)
{
  o_task*               from;
  o_task*               to;
  o_as*                 o;

  [...]
}
```

Structures and union fields must be aligned with the type name, using tabulations. In addition, when declaring a structure or union type, there must be only one field declaration per line.

```
typedef struct
{
  i_event                       eventid;

  t_type                        type;

  u_event_handler               handler;
}                               o_event;
```

Enumerations values must be capitalized and must appear on their own line.

When an expression, declaration, assignment *etc.* spans over multiple lines, the additional lines must be indented according to the type of statement. It is indeed the responsibility of the developer to align these lines properly. Note that *Emacs*'s alignment comply to many of the kaneton rules.

```
int           x = y * foo(z + pow(z, 3)) +
  a_very_very_long_function_name(y, z) + bar(baz(42, 21), y * z);
```

Declarations

There must be only one declaration per line.

External declarations must not be located in functions blocks but rather in the global scope.

Variables may be initialized at the point of declaration. This way, bugs detection could be improved. For this purpose, however, valid expressions are only those composed of constants and macros.

Below are illustrated some very bad constructs:

```
t_error                 task_current(i_task*                    tsk)
{
  i_thread              current, next;
  o_thread*             o = thread_get(current);
  extern int            sched;

  [...]
}
```

Variables must always be declared at the beginning of a block.

Statements

A single line must not contain more than one statement. In addition, commas must not be used on a line to separate statements.

The comma must be followed by a single space, except when they separate arguments in functions or macro functions declarations and calls, and the argument list spans multiple lines.

The semicolon must be followed by a newline and must not be preceded by a whitespace, except if alone on its line.

```
{
  int                   i;

  for (i = 0; i < 256; i++)
    ;

  [...]

  obscure_example(id, as);

  [...]
}
```

There exist exception to the above rules. For more information, please refer to the subsection about *Control Structures*.

Statement keywords must be followed by a single whitespace, except those without arguments. This especially implies the `return` without argument, like `continue` and `break`, must not be separated from the following semicolon.

Statement keywords which take an argument must enclose the argument between parentheses, as illustrated next.

```
{
  i_timer               id;

  while (1)
    {
      if (id != ID_UNUSED)
        continue;

      [...]

      return (0);
    }
}
```

Finally, the use of the `goto` statement must be extremely limited.

Expressions

All binary and ternary operators must be padded on the left and right by one space, including assignment operators.

Prefix and suffix operators must not be padded neither on the left nor on the right.

When necessary, padding is always done with a single whitespace.

The `.` and `->` operators must no be padded.

Below is an example illustrating these rules.

```
{
  int*        p;

  x = 10 * *p + reference->value++;

  x += ((*reference).tag == 1 ? 10 : 0);
}
```

There must not be any whitespaces between the function name and the opening parenthesis in function calls.

Expressions may span over multiple lines. When a line break occurs within an expression, it must appear just after a binary operator, in which case the binary operator must not be padded on the right by a whitespace.

6.4.4 Control Structures

General

Control structure keywords must be followed by a whitespace. The conditional parts of algorithmic constructs - `if`, `while`, `do`, `for`, `else` - must be alone on their line.

The following constructs are incorrect:

```
{
  if(option & OPTION_SET)
    return 0;

  while (str[i]) write(1, str[i++], 1)

  if (id != ID_UNUSED) {
    foo();
  } else {
    bar();
  }

  do {
    ++x;
  } while (x < 10);
}
```

The following must be prefered:

```
{
  if (option & OPTION_SET)
    return (0);

  while (str[i])
    write(1, str[i++], 1)

  if (id != ID_UNUSED)
    {
      foo();
    }
  else
    {
      bar();
    }

  do
    {
      ++x;
    } while (x < 10);
}
```

Finally, the conditional control structures must always use a full comparison and avoid the use of the logical not operator: `!`

For instance, the following construct is very wrong:

```
if (!(kernel_buffer = malloc(size)))
  MESSAGE_LEAVE(message, ERROR_UNKNOWN);
```

Indeed, this construct supposes the `malloc()` function returns `0` when it fails. However, the `malloc()` function returns `NULL` when it fails and no specification actually indicates what is the value of the `NULL` macro. Many programmers consider this macro is always equivalent to the `zero` integer value but this is not true.

Therefore, kaneton developers must use comparison operators to avoid such issues. In addition, using comparison operators makes the code much more simple to read and understand:

```
if ((kernel_buffer = malloc(size)) == NULL)
  MESSAGE_LEAVE(message, ERROR_UNKNOWN);
```

for

As a general exception, the `for` construct breaks many of the previously defined rules.

Multiple statements may appear in the initial and iteration part of the `for` structure. For this effect, commas must be used to separate statements.

The three part of the `for` construct may span over multiple lines. The `while` construct must be prefered to a `for` construct with three empty parts.

The following examples are very wrong:

```
{
  int         i;

  for (i = 0, int j = 1;
       j = j + 2, i < 10;
```

```
        i++)
    ;

  for (;;) ;
}
```

Instead, prefer:

```
{
  int         i;
  int         j;

  for (i = 0, j = 1, j = j + 2;
       i < 10;
       j = j + 2, i++)
    ;

  for (; 1; )
    ;
}
```

Finally, single-line loops - `for` and `while` - must have their terminating semicolon on the following line, as illustrated above.

6.4.5 kaneton

The following rules apply specifically to the source code of the microkernel itself.

In kaneton, function declarations follow the same template. Indeed, the return value must always indicate whether the function call successed or not. The return type is therefore always the same: `t_error`.

Function arguments must be ordered, starting with input argument and finishing with ouput arguments. Moreover, the first argument must be the more important *i.e.* the kaneton object identifier or capability on which the action is performed.

```
t_error                 task_reserve(t_class                    class,
                                     t_behav                    behav,
                                     t_prior                    prior,
                                     i_task*                    id);
```

In this example, `class`, `behav` and `prior` are input arguments. The function creates a new task object and returns the identifier of this new task into the `id` output argument, which is the last of the list.

Especially note that kaneton does not have any function returning a value. The return value is always used as an error status, thus leading to a more uniform and coherent programming style.

Functions located in a kaneton manager must start with a call to the `ENTER()` macro function and finish with a call to the `LEAVE()` macro function. These macro functions were introduced in order to facilitate the process of adding a statement at the beginning and/or at the end of a function. Especially, the `return` keyword should no longer be used.

Below is an example of such calls:

```
/*
 * this function releases a thread from a given task.
 *
 * steps:
 *
 * 1) call the machine-dependent code.
 * 2) get the thread object.
 * 3) get the task object.
 * 4) release the thread stack if needed.
 * 5) release the thread-s object identifer.
 * 6) remove the thread from the task threads list.
 * 7) remove the thread from the threads set.
 */

t_error                 thread_release(i_thread                 threadid)
{
  o_task*               task;
  o_thread*             o;

  THREAD_ENTER(thread);

  /*
   * 1)
   */

  if (machine_call(thread, thread_release, threadid) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  /*
   * 2)
   */

  if (thread_get(threadid, &o) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  /*
   * 3)
   */

  if (task_get(o->taskid, &task) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  /*
   * 4)
   */

  if (o->stack)
    {
      if (map_release(task->asid, o->stack) != ERROR_NONE)
        THREAD_LEAVE(thread, ERROR_UNKNOWN);
    }

  /*
   * 5)
   */

  if (id_release(&thread->id, o->threadid) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  /*
   * 6)
   */

  if (set_remove(task->threads, threadid) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  /*
```

```
   * 7)
   */

  if (set_remove(thread->threads, threadid) != ERROR_NONE)
    THREAD_LEAVE(thread, ERROR_UNKNOWN);

  THREAD_LEAVE(thread, ERROR_NONE);
}
```

In this example, the types of error are not distinguished as the function always returns `ERROR_UNKNOWN` which is not a very good idea.

6.5 LaTeX

The kaneton documents are written in the LaTeX language. These documents are viewable through the *view* tool. For more information about this tool, please refer to the *Section 5.1.3*.

6.5.1 Naming

LaTeX files must be named in lower-case letters. Moreover, composite file names must be separated by a dash `-`.

6.5.2 Layout

Since LaTeX is not a very readable language, every contributor is asked to follow the rules described in order to make the documents internal representation looks like the resulted output.

Indeed, writers are asked not to use LaTeX commands specifying layout requirements especially about indentation like `\paragraph`, `\vspace` *etc.* Since the kaneton project provides templates, every kaneton document must rely on a template which specifies the paragraph indentation, paragraph space *etc.*

Thus, the writer willing to distinguish two paragraphs in his document should simply put a single blank line between to two texts in his LaTeX file as shown below:

```
This is a paragraph which is intended to explain nothing special but
how to construct paragraphs in a very simple and readable way.

Then, after a single blank line, this text will be considered as a new
paragraph.
```

This rule aims at making the document code as readable as possible. However, if a vertical space is needed in some place the default paragraph indentation does not apply, then, the `\-`LaTeX command must be used. This is especially useful in *lecture* documents.

The example below illustrates such a *lecture* document and the need of vertical indentation:

```
\begin{frame}
  \frametitle{Description}

  The MIPS processor is a 32-bit little-endian processor.
```

```
\-

This processor provides \textbf{32 integer registers}, from R0 to R31.

[...]
```

Obvisouly, lines must not exceed 80 characters in width.

The templates provided by kaneton are located in the `view/template/` directory. Those include *book*, *paper*, *exam* *etc.*

Therefore, every document should start by including the template file. However, since the template files need to access other files like LaTeX dependency files, additional packages *etc.* these templates need to know where the document directory is located from the root `view/` directory. Every document must therefore specify the path to the `view/` directory before including the template file.

Every file should then start with a *setup* section similar to the following:

```
%
% ------- setup ----------------------------------------
%

%
% path
%

\def\path{../..}

%
% template
%

\input{\path/template/book.tex}
```

As every other type of kaneton file, the LaTeX files are composed of sections. Below are listed some of possible sections but note that it highly depends on the organisation of the document: multiple files, *etc.* as well as the type of document: presentation slides, paper, book *etc.*

The best way to make things properly is to look at the existing documents.

- **header**: this section contains the file header which provides information on the file edition: creation, last update *etc.*
- **setup**: this section contains the LaTeX setup: path to the `view/` directory, template including, title definition *etc.*

```
%
% ------- setup ----------------------------------
%

%
% path
%

\def\path{../..}

%
% template
%
```

```
\input{\path/template/book.tex}

%
% header
%

\lhead{\scriptsize{The kaneton microkernel :: development}}

%
% title
%

\title{The kaneton microkernel :: development
       \version
       \logo}
```

- **text**: this section is used when the text is relatively short and likely to fit in a single file. The section then contains the whole document's text.

  Instead, a section whose name is based on the current chapter or section can be used if the document is much more larger and generally split into multiple files.

```
%
% ------- latex ------------------------------------
%

\section{\LaTeX}

The kaneton documents are written [...]
```

6.5.3 Style

First of all, comments must be written in lower-case letters. Comments are not intended, in LaTeX files, to describe what the file do but instead play the role of visual separators.

Note that documents are either *public* or *private*. Indeed, the LaTeX definition `\mode`, stored in a temporary file included by kaneton templates, can be used to hide information which must be kept private to the kaneton developers community like, for instance, implementation details. For more information, take a look at the template you are interested in or at the other equivalent kaneton documents.

Recall that the `view/figures/` directory contains figures related to the kaneton microkernel documents. These figures must always be prefered to specific figures. Moreover, writers are asked to put their figures in this directory if the figure is general enough. Furthermore, figures must be in the *FIG* format. Finally, figures must be exported into the *PDF* format as it is the only format which is accepted both by pure LaTeX documents as well as *Beamer* documents while rendering without quality loss.

Remember that every element must be aligned according to its parent. For instance, in the *enumerate* environment, the text related to the item must be placed on a new line, aligned by two characters from the `\item` element as shown below.

```
\begin{itemize}[<+->]
  \item
    \textbf{R2}: return value.
  \item
```

```
    \textbf{R29}: frame pointer.
  \item
    \textbf{R30}: global variables area pointer.
\end{itemize}
```

Templates

The *kaneton* development environment provides templates in order to making writing documents easier. The provided templates can be grouped into two categories depending on the type of rendering: **article**s are pure LaTeX documents while **presentation**s are *Beamer*-based slides.

The templates *book*, *exam*, *feedback*, *internship* and *paper* belong to the *article* category while the templates *lecture* and *talk* are *presentation*s.

The remaining of this section is details the style according to the document category.

Article

A long comment separator, composed of three lines, must be used before every new subsections, while a single line separator must be used for subsubsections and other less important components.

```
%
% naming
%

\subsection{Naming}

Names in kaneton must comply to the following rules.

% general

\subsubsection{General}
```

Nevertheless, this rule also depends on the document organisation.

Presentation

Unlike *article*s, *presentation*s are based on the *Beamer* package. These documents are therefore composed of slides — frames in the *Beamer* terminology.

Each frame must be preceded by a commented number indicating the slide's number into the section of subsection.

```
%
% introduction
%

\section{Introduction}

% 1)

\begin{frame}
  \frametitle{Description}

  \begin{itemize}[<+->]
    \item
      About \textbf{thirty} hours course.
    \item
      Concluded by an exam.
  \end{itemize}
```

```
\end{frame}

% 2)

\begin{frame}
  \frametitle{Contents}

  \begin{itemize}[<+->]
    \item
      External architecture.
    \item
      Pipeline.
    \item
      Compiler optimisations.
    \item
      Memory.
  \end{itemize}
\end{frame}
```

opk Package

Since every template is likely to need providing writers the same set of functionalities, a LaTeX package has been written. The **opk** package includes a set of commands useful for building function definitions, referencing LaTeX labels *etc.* This package is located in `view/package/opk/` and is composed of the functions below.

First, needless to define the `\author` directive as the *opk* package does it automatically, unless the writer knows what he is doing. Additionally, the package redefines the *verbatim* environment.

`\term`(`text`)

This function is used for introducing new terms.

`\name`(`text`)

This function is used for referering to already introduced names.

`\code`(`text`)

This function is used for words that represent function names or anything related to source code.

`\reference`(`section/figure/etc.`)

This command is used for referencing figures, sections *etc.*.

Note that this command should encapsulate the whole text such as: `\reference{Figure \ref{figure:Experiments}}`.

`\location`(`location`)

This command is used for describing locations: path, *URL etc.*

`\function`(`return type`, `function name`, `arguments list`, `description text`)

This special command is used for describing function definitions.

\type(argument)

This function is for describing arguments.

This command should only be used in the \function() command.

\command(command line, description text)

This function is equivalent to the \function() one but targets *Shell*, *Python* *etc.* commands rather than language functions.

\subsubsubsection(section name)

This weird command is just an easier way of sub-dividing the text, once more.

\question(text, answer length)

This command is used in *feedback*-template document for creating questions.

\latex(command name)

This command "latexify" the given command name by putting a backslash in front of it.

\note(text)

This command is used for making little notes, tips, hints *etc.*

\example(text)

This command is used for creating examples.

\ie()

This command genereates *i.e.*.

\eg()

This command genereates *e.g.*.

\etc()

This command genereates *etc.*.

\aka()

This command genereates *a.k.a.*.

Plus, layout-oriented commands are also provided.

\indentation()

This command initialises the indentation for the document.

\logo()

This command displays the *kaneton* logo.

\version()

This command displays the version type of this document according to the \mode definition.

In addition, specific LaTeX environments are provided.

\details()

The *details* environment enables writers to include additional information which will only be viewable to private members *i.e.* in the private version of the document.

\correction()

The *correction* environment plays the same role as the *details* environment but is used in *exam*-template documents for providing exercises' answer.

Chapter 7

People

This sections lists the people in charge of the different tools.

7.1 Project

Below are listed the different components of the kaneton project and, for each, the person in charge of it.

- **Development Environment**: *Julien Quintard*;
- **Design**: *Julien Quintard*;
- **Core**: *Julien Quintard & Matthieu Bucchianeri*;
- **Architecture IA-32**: *Matthieu Bucchianeri*;
- **Architecture MIPS-64**: *Enguerrand Raymond*;

7.2 Tools

As explained in the documents, newcomers must first register to be able to use a number of tools. In order to finalize registrations and to perform them, new contributors must get in touch with the people responsible of these tools.

Below are listed the persons in charge of each tool whose usage is restricted.

- **Mailing-List**: *Julien Quintard*;
- **Repository**: *Julien Quintard*;
- **Wiki**: *Julien Quintard*;
- **Project Management Tool**: *Julien Quintard*;
- **Test**: *Julien Quintard*;
- **Build Farm**: *Julien Quintard*;
- **Export**: *Francois Goudal*.

Chapter 8

Licenses

In this chapter are described the licenses in relation with the kaneton project.

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 microkernel project is under the *kaneton* license.

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

The sections below 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.