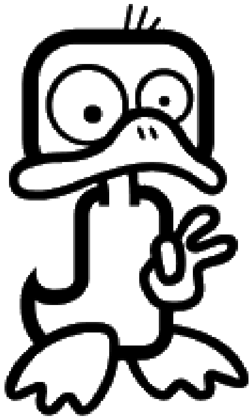
The kaneton microkernel :: assignments



kaneton people

February 11, 2014

This document contains everything necessary for students to undertake the kaneton educational project.

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

---

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

Contents

Chapter 1

Introduction

This chapter briefly introduces the purpose of this documentation.

The *kaneton* educational project enables students to develop their own micro-kernel as a way of understanding operating systems internals.

As anyone can imagine, such a project takes a huge amount of time and motivation. While the motivation will anyway play an important role in the success of students' project, the time spent can be greatly reduced if students focus on implementing some specific parts rather than developing a complete micro-kernel from scratch.

Indeed, as we will see later in this document, the current *kaneton* educational project comes with a student **snapshot** which contains a complete development environment as well as the source code skeleton of the kernel.

Some people would probably prefer working on their own micro-kernel design and implementation, starting from scratch. All we can wish to such people is enough motivation to keep working on their project long enough to be satisfied, luck and hard work.

Either way, going through the *kaneton* micro-kernel documentation should not be a waste of time. Especially, people interested in developing their own project from scratch could take a look at the *kaneton* design in case they like it enough to implement it their way.

The remaining of this document is organised as follows. *Chapter 2* defines what students should be familiar with beforehand. *Chapter 3* details the multiple ways for students to get help throughout the project. *Chapter 4* briefly presents the student snapshot. Then, *Chapter 5* presents the first project stage which takes place outside the microkernel while *Chapter 6*, *Chapter 7* and *Chapter 8* detail the assignments of the following stages. Finally, *Chapter 9* discusses what students could do after having undertaken such a project.

Chapter 2

Requirements

This chapter discusses the requirements for students to undertake the *kaneton* educational project.

The requirements can be divided into two categories: **theoretical** and **practical**. Any student lacking one of the following points should first make sure he/she has a good knowledge of it before starting the project.

Note however that students can learn everything along the project though we recommend a minimum of understanding regarding the operating system principles and programming languages.

The *Opaak Wiki* accessible at `http://kaneton.opaak.org/wiki` contains resources and references which could be useful to the student willing to document himself.

2.1 Theoretical

The following topics must be known from every student willing to undertake the project:

- **Operating Systems Principles** and especially micro-kernel architectures;
- **Computer Architectures**.

2.2 Practical

Since the goal of the project remains to develop a micro-kernel, every student should be familiar with the programming languages and development environments the project relies on:

- **UNIX** environment or your operating system environment — as long as it is supported by the kaneton development environment;
- **C** programming language;
- **Assembly** programming language.

Chapter 3

Support

This chapter discusses the means for students to get help.

Programming a complete micro-kernel can be very hard for your nerves, and it is sometimes just as easy to ask someone for advise, opinion, tips *etc.*

The *kaneton* people therefore decided to provide students a number of tools for students to seek information.

3.1 Mailing-List

The mailing-list can be used by students for both communicating with other students and *kaneton* people.

Note that rules must be followed when it comes to using this communication medium. Indeed, people should be careful regarding the topics addressed on this mailing-list especially when it comes to giving too much details related to a specific implementation problem.

Since other students are also available on this mailing-list, you should not hesitate to ask for help as they probably ran into the same problem at the time they were working on this part. If not, the kaneton teachers will always try to answer your questions as best as they can.

The mailing-list can be accessed at the following email address:

`kaneton@googlegroups.com`

Subscribe by sending an email to:

`kaneton+subscribe@googlegroups.com`

3.2 Wiki

The *Wiki* enables students to access information useful to the *kaneton* educational project. Note that students should consider it at their information space and are obviously welcome to suggest content which should be added or even modified.

The *kaneton Wiki* can be accessed at the following address:

`http://kaneton.opaak.org/wiki`

Chapter 4

Snapshot

This chapter introduces the snapshot which provides students everything necessary for starting the development of the *kaneton* microkernel.

Although *k0* is a standalone project, the following stages, namely, *k1*, *k2* and *k3* take place within the microkernel snapshot. In addition, students require the educational snapshot in order to submit their implementation. Thus, the student should download the educational snapshot from the kaneton website `http://kaneton.opaak.org`. Once downloaded, students should read the dedicated article on the *Opaak Wiki* and follow the steps in order to configure their kaneton environment.

Note that in order for both the testing and submission processes to work properly, students must comply with the following rules:

- The test and submission procedures rely on a mechanism which essentially consists in archiving the whole kaneton development directoy, cleaning it from the object files and versoning directories such as `.svn/`, `.git/` *etc.* before compressing it and sending it to the server.

  Once on the server, the snapshot is installed on a small disk image and passed to a specific virtual machine whose purpose is to compile the kaneton microkernel, generating a bootable image. Thus, the snapshot provided to the server should be small enough, around 400KB, in order to fit on the disk image.

  Students must therefore make sure to avoid placing needless data in the development directory such as architecture manuals, copies of the whole source code etc. Students can check the snapshot size of their current development environment through the following command:

```
$> cd kaneton/export/
$kaneton/export> make export-test:group
[...]
$kaneton/export> ls -l output/
total 404K
-rw------- 1 mycure mycure 398K Mar  7 13:40 test:group.tar.bz2
$kaneton/export>
```

  In this example, the snapshot has a size of 398KB. Note that every snapshot exceeding the size of 5MB will for sure generate an error once tested or evaluted.

- The *test* user profile located in `environment/profile/user/` should never be removed as it is used by the testing and evaluation procedures.

- The test and submission systems rely on *Python*—more precisely has been developed with *Python 2.6*—and thus requires packages which students must install, including *yaml*, *hmac*, *pickle* and *xmlrpc*.

Finally, note that a test capability must first be acquired in order to be able to use the test and evaluation system. The following assumes that students possess such a capability. If not, he or he should contact the kaneton contributors by email at:

`admin@opaak.org`

4.1 Test

The kaneton test system enables students to test their implementation against a set of test suites. However, the number of test suites students are allowed to request is limited. As such, the kaneton test system should be seen more as a validation system. Students are therefore invited to implement their own test suite in order to thorougly test their kaneton implementation.

The following command shows how to retrieve information regarding the student test profile.

```
$> cd test/client/
$test/client> make information
[+] configuration:
[+]   server:                 https://test.opaak.org:8421
[+]   capability:             /home/chiche/kaneton/environment/profile/user/chiche/chiche.cap
[+]   platform:               ibm-pc
[+]   architecture:           ia32/educational

[+] information:
[+]   profile:
[+]     attributes:
[+]     identifier:           chiche
[+]     type:                 student
[+]     members:
[+]       name:               Florent Chichery
[+]       email:              chiche@epita.fr
[+]   suites:
[+]     k3:                   This test suite contains tests related to the execution.
[+]     k2:                   This test suite focuses on the memory management.
[+]     k1:                   This test suite focuses on the event processing.
[+]   stages:
[+]     k3:                   This stage evaluates the kaneton's execution functionalities.
[+]     k2:                   This stage evaluates the kaneton's memory management.
[+]     k1:                   This stage evaluates the kaneton's event procesing capabilities.
[+]   environments:
[+]     xen:                  The 'xen' environment is used to thoroughly test a kaneton implementation in a
[+]     qemu:                 The 'qemu' environment is used to test a kaneton implementation through the QEM
[+]   database:
[+]     reports:
[+]       xen:
[+]         ibm-pc.ia32/educational:
[+]           k3:
[+]                           20110206:113544
[+]                           20110206:114823
[+]           k2:
[+]                           20110205:222312
[+]           k1:
[+]                           20110205:211847
[+]                           20110205:213502
[+]                           20110205:214408
[+]     quotas:
[+]       xen:
[+]         ibm-pc.ia32/educational:
[+]           k3:             3
[+]           k2:             3
[+]           k1:             3
[+]   deliveries:
[+]     k1:                   None
[+]     k2:                   None
[+]     k3:                   None
$test/client>
```

Finally, the following command illustrates a student triggering a test suite on the *Xen* environment.

```
$test/client> make test-xen::k1
[+] configuration:
[+]   server:                 https://test.opaak.org:8421
[+]   capability:             /home/chiche/kaneton/environment/profile/user/chiche/chiche.cap
[+]   platform:               ibm-pc
[+]   architecture:           ia32/educational

[+] generating the kaneton snapshot
[+] loading the kaneton snapshot
[+] requesting the server
[+] the snapshot has been scheduled for testing under the identifier: 20110208:175055
```

```
$test/client>
```

Once the testing complete, the student will receive an email containing a detailed report.

4.2 Submission

In order to submit an implementation for evaluation, the student simply has to issue the *submit* command, as shown below.

```
$test/client> make submit-k2
[+] configuration:
[+]   server:                 https://test.opaak.org:8421
[+]   capability:             /home/chiche/kaneton/environment/profile/user/chiche/chiche.cap
[+]   platform:               ibm-pc
[+]   architecture:           ia32/educational

[+] generating the kaneton snapshot
[+] loading the kaneton snapshot
[+] requesting the server
[+] the snapshot has been submitted successfully
$test/client>
```

Note that any submitted snapshot must comply with the rules related to the testing procedure since the evaluation process is essentially the same.

4.3 Cheating

Students should already be familiar with the fact that cheating is strictly prohibited. As such, should the implementation of a function, an algorithm or even a test look too similar to another student's, the teachers will be forced to take measures ranging from a -42 grade for the current stage to the definitive exclusion of the student from the project.

Chapter 5

k0

k0 is an introduction to low-level programming through the development of a simplistic boot loader. Note that this stage is not critical because the following stages will not rely on this boot loader as *GRUB - GRand Unified Bootloader* will be used instead.

Through this stage, the student will manipulate several concepts including:

1. **Computer Boot Process**

   The several phases a computer runs through in order to check the machine's state, to the location of the bootable devices until the execution is transferred to the first valid boot sector found.

2. **Assembly Language**

   The boot loader will be entirely developed in the Intel *x86* assembly language.

3. ***BIOS - Basic Input/Output System* Interrupts**

   The *BIOS* provides basic functionalities for reading the floppy, displaying text to the screen *etc.* These functionalities will be used in the first exercices to get used to the low-level development environment.

4. **Processor Modes**

   The *Intel* microprocessor can run in different modes depending on the word length—16 or 32 bits—, the protection capabilities *etc.*

   While the simplest mode will be used at first, enabling students to use the *BIOS*' features, a more evolved mode will finally be used as required before handing the execution to the kernel *i.e.* the *Protected Mode*.

5. ***ELF - Executable and Linking Format***

   The boot loader, written in assembly, is packed in a raw format to be detected as a valid boot loader by the computer's *BIOS*.

   However, the kernel, probably written in *C* or *C++*, will be packed in an advanced format referred to as *ELF*. Although this format has numerous capabilities, the boot loader will basically find the entry point and jump on it.

5.1 Development

Students are advised to use *GCC - GNU Compiler Collection* to assemble their source files and *GNU LD* to link them into binary files. Note that students must therefore follow the *GAS -GNU Assembler* syntax.

The following example illustrates this two-step process:

```
$> gcc -c bootsect.S -o bootsect.o
$> ld bootsect.o -o bootsect --oformat binary -Ttext 0x7c00
```

In order to execute your code, students should use the *QEMU* emulator, as follows:

```
$> qemu-system-i386 -fda bootsect
```

Let us recall that *x86* microprocessors start executing in 16-bit. A directive must therefore be used to specify the assembler that the code must be generated in 16-bit. Note that students will later be asked to switch to a 32-bit mode referred to as the *Protected Mode*. As such, another directive will have to be used, hence mixing both 16- and 32-bit code.

5.2 Resources

Students will find additional resources related to the *k0* stage on the *Wiki* at `http://kaneton.opaak.org/wiki`.

5.3 Test

Students will need to test the functions they implement. In order to keep the test environment from the exercise files, it is advise to rely on the *CPP - C Pre-Processor* for including the exercises' functions within the test file.

The following example illustrates a test file which includes the first exercise's file:

```
.globl _start

_start:
  mov 0x42, %al
  call print_char

loop:
  jmp loop

#include "ex1.S"
```

This way, the test code is always kept apart from the exercises so that the risk for students to submit their tests is limited.

5.4 Submission

Every student is expected to submit a snapshot containing the following structure. Note that the evaluation system is automatic. Therefore, any snapshot which does not conform with this structure will obviously be assigned the grade of zero:

```
ex1/
  ex1.S
ex2/
  ex2.S
ex3/
  ex3.S
ex4/
  ex4.S
ex5/
  ex5.S
ex6/
  ex6.S
```

The submitted snapshot should therefore contain nothing more than these six directories and files. Please be careful with the extension of the source files; the 'S' is uppercase!

The *k0* stage is composed of several exercises. For each exercise, the student is asked to write functions in assembly language taking arguments in some registers and providing output in others. Note that these functions should take care not to modify any register unless explicitly stated.

Every exercise is attached to a specific file, `ex2.S` for instance. Such a file must therefore contain the labels of the functions that the student is supposed to provide.

In addition, every file should end with the bootsector signature, making the functions of the file useable within a boot loader. This signature, composed of the bytes `0xaa55` must be located at the 511th and 512th byte of the first boot sector. Thus, the boot sector must be filled up until the 511th byte is reached, at which position the signature must be put.

The example below illustrates such an exercise file, containing two labels and the bootsector signature:

```
  .code16

  requested_function_1:
    mov $0x42, %ah
    mov %bl, %bl

  local_label:
    int $0x10
    xor %dh, %dh
    jmp local_label
    ret

signature:
  .org 510, 0
  .short 0xaa55
```

Some exercises will require students to output text on the screen. Bear in mind that exercises are evaluated automatically. Therefore, color is not taken into account. Besides, students should never try to clear the screen of the text printed by the *BIOS* or the emulator. By doing so, the evaluation system would catch the sequence and consider it as text being printed by your functions.

Finally, as explained previously, students will need to test their functions in order to make sure they behave properly. Students must remember to remove these tests from the submitted snapshot. Thus, your source files should never include a label `_start` or any label starting with `__`. Note however that the bootsector signatures must remain.

Note that this stage differs slightly from the following ones regarding the submission. Indeed, in the following stages, which take place within the microkernel, the test client automatically packages the student's current implementation and sends it to the server.

For this stage however, the student is asked to package its source files manually, according to the hierarchical structure given above. Then, the freshly created snapshot must be placed in the client directory under the name `snapshot.tar.bz2` *i.e.* at the precise location: `test/client/snapshot.tar.bz2`.

Finally, the test client can be used to submit the stage *k0*. The client, noticing the presence of the snapshot, will use it instead of generating a snapshot from the current kaneton implementation.

The lines below illustrates this process.

```
$k0> ls
ex1/ ex2/ ex3/ ex4/ ex5/ ex6/
$k0> tar cjvf k0.tar.bz2 *
[...]
$k0> mv k0.tar.bz2 ~/kaneton/test/client/snapshot.tar.bz2
$k0> cd ~/kaneton/test/client/
$~/kaneton/test/client> ls
client.py Makefile README snapshot.tar.bz2
$~/kaneton/test/client> make submit-k0
[+] configuration:
[+]   server:                https://test.opaak.org:8421
[+]   capability:            /home/chiche/kaneton/environment/profile/user/florent.chichery/florent.chichery
[+]   platform:              ibm-pc
[+]   architecture:          ia32/educational

[+] loading the snapshot '/home/chiche/kaneton/test/client/snapshot.tar.bz2'
[+] requesting the server
[+] the snapshot has been submitted successfully
$~/kaneton/test/client>
```

Note that once the snapshot has been submitted, students should remove the file from the `test/client/` directory in order to prevent it from being used in the other stages.

5.5 Evaluation

The student's code will be evaluated by testing every function automatically.

The evaluation process will therefore go as follows. First, the student exercise file will be linked with an evaluation file containing the entry point. This main function will then trigger some function calls and verify that these functions behave according to the specifications defined in this document.

Let us recall that the whole binary—composed of the student exercise file and the evaluation function—must fit in a bootsector *i.e.* 512 bytes.

Therefore, for every exercise, the size of the evaluation code is given as an indication. As such, students should make sure the size of their code does not exceed the remaining capacity of a bootsector.

In order to make sure a student's exercise does not exceed this capacity, the following directive can be used:

```
.space 42,0

#include "ex1.S"
```

Note that the value 42 will have to be replaced by the size of the evaluation code given in each exercise.

If the assembler refuses to generate a binary given a test file containing this directive, this would mean that the exercise code takes too much space, in which case the student will have to optimise it. The error the assembler returns in such scenarios is similar to the one below:

```
ex1.S: Assembler messages:
ex1.S:12: Error: attempt to move .org backwards
```

5.6 Exercises

This section discusses the various exercises contained in this stage.

5.6.1 Console

Through this exercise the student will learn how to use *BIOS* services in order to print strings to the console.

File	**Space**
`ex1/ex1.S`	175 bytes

This exercise is divided into several steps through the implementation of the following functions.

`print_char`

This function prints the character whose *ASCII* code is given in `%al`.

Note that the character is displayed at current cursor's position.

`cursor_set`

This function set the cursor's position at row `%dh` and column `%dl`.

`print_string`

Finally, this function prints the string referred to by `%si` at the position given by row `%dh` and column `%dl`.

5.6.2 Registers

In this exercise, students are invited to implement several functions well-known from *POSIX* programmers. These functions will finally be used to provide a function used to dump the state of the 16-bit registers.

File	Space
`ex2/ex2.S`	155 bytes

`malloc`

This function allocates `%ax` bytes of memory and returns the address in `%di`.

First, the base address of the head should be statically declared. Then, whenever the function is called, the next available address is used and incremented.

`itoa_hex`

This function converts the integer in `%ax` into a string using the hexadecimal representation, the address of the string being returned in `%si`

Note that the hexadecimal format must comply with the regular expression `0x[0-9a-f]{4}`. For example, the number 42 should be converted to the string `'0x002a'`.

This function could for instance be used by students to check the address returned by the `malloc()` function.

`dump_registers`

This function dumps the values of the registers *ax*, *bx*, *cx*, *dx*, *si* and *di*.

This function must start displaying at the current cursor's position. Besides, the evaluation system expects the function to come back to the first line when the end of the screen is reached.

In essence, the output should be exactly identical to the one below:

```
ax = 0x1234
bx = 0x0000
cx = 0xabcd
dx = 0x00ff
si = 0x1000
di = 0x0ff8
```

5.6.3 Keyboard

In this exercise, the aim is to implement a basic keyboard driver by relying on the *BIOS* services.

File	Space
`ex3/ex3.S`	155 bytes

`get_key`

This function returns in `%al` the *ASCII* code of the key which has been pressed.

`getln`

This function reads and prints every character received from the keyboard. When *ENTER* is entered however, a *newline* is printed and the function returns.

Note that none of the modifiers such as *SHIFT*, *ALT*, *CTRL*, *CAPS LOCK etc.* will be tested. Likewise, the arrows, *BACKSPACE etc.* will also be ignored.

Finally, the end of line does not have to be handled as it will not be tested.

5.6.4 Floppy

Further *BIOS* interrupts are introduced here to make the $16 - bit$ programmer able to control the floopy drive. The objective of this exercise is to read the second sector from the floppy drive in order to check whether it is a valid bootsector.

Note that the second sector is considered because the first has already been loaded into memory. This technique is used by boot loaders, such as *GRUB*, to provide *chain loading*.

File	**Space**
`ex4/ex4.S`	155 bytes

`floppy_init`

This function initializes the floppy drive.

`is_bootsector`

This function calls the initialization of the floppy drive, loads the second sector into memory and checks that it is a valid bootsector.

Noteworthy is that sectors are numbered starting with 1, not 0.

Once the sector checked, the function prints a string, along with the read boot signature. Note that the function must prints at the current cursor's position.

The following illustrates both cases, respectively a valid and invalid bootsector.

```
magic found: 0xaa55
wrong magic: 0x0042
```

5.6.5 Mode

Until now, the code students have been developing were written in 16-bit, hence evolving in *Real Mode*. This mode is a relique of the past. As most kernels are written in 32-bit, the boot loader must prepare the environment to fit the needs of the kernel.

Through this exercise, the students are going to switch from the *Real Mode* to the *Protected Mode*. This process requires the developer to set up some hardware data structures indicating the microprocessor how it must behave. The data structure of interest if the *GDT* - *Global Descriptor Table*.

File	**Space**
`ex5/ex5.S`	225 bytes

Students should read the necessary material, especially the *Intel* manual *Volume 3A* which contains a chapter on the *Protected Mode*.

The following describes the steps to follow to perform a switch to the *Protected Mode*:

1. Create and fill the *GDT* with the necessary entries;
2. Disable the interrupts through the `cli` instruction which masks the hardware maskable interrupts;
3. Set the *PE* flag in the control register *CR0*;
4. Immediately perform a long jump in order to change the *CS* - *Code Segment* selector, referencing the appropriate *GDT* entry;
5. Update the other segment selectors *DS*, *ES*, *FS*, *GS* and *SS*.

Do not forget that the code executed after switching from the *Real Mode* to the *Protected Mode* must be assembled in 32-bit.

In addition, *BIOS* interrupts cannot be used in *Protected Mode*. Therefore, the basic functionalities must be re-implemented in 32-bit and without the support of the *BIOS* services.

`pmode_switch`

This function switches from the *Real Mode* to the *Protected Mode*.

Note that when calling this function, the *CPU* will push the return address as a 16-bit value, since operating in *Real Mode*. However, when returning, the *CPU* will pop a 32-bit value since now operating in *Protected Mode*.

The student should not worry about this detail when implementing this function. Indeed, it is the responsability of the caller to push another 16-bit zero on the stack before calling so that when pop-ed in 32-bit both values coincide.

`print_string32`

This function prints the string whose address is given in `%esi`.

The location is specified through the row by `%ecx` and the column by `%edx`. Note that the row and column are starting at index 0. As such, the upper left corner has coordinates $(0, 0)$.

Finally, the student should ignore the end of line as this will not be tested. Besides, colors can be changed but will not be considered by the evaluation system. Moreover, the special characters such as `'\r'`, `'\t'` *etc.* will not be tested either.

5.6.6 ELF

The final exercise of the *k0* stage consists in writing the core functionality of any boot loader: loading the kernel binary from a device into memory and executing it.

Since most kernels are written in high-level languages such as *C* or *C++*, such binaries are actually packaged according to a flexible format, the most common one on *UNIX* systems being the *ELF - Executable and Linking Format*.

File	Space
`ex6/ex6.S`	145 bytes

Note that an *ELF* example binary is provided on the *Wiki*. As mentioned earlier, boot loaders are usually located on the first sector of the boot image being on a floppy or hard drive. Then follows, on one or more sectors the kernel, the role of the boot loader being to load the kernel sectors and execute it.

The following command shows how to create a bootable image given a boot loader binary and an *ELF* kernel:

```
$> cat bootsector kernel.elf /dev/zero | dd bs=512 count=2880 of=disk.img
```

The *ELF* is a complex format composed of headers, segments and tables referencing them. The kernel binary will therefore have to be parsed in order to extract the code and its entry point on which the boot loader will have to jump to transfer the execution flow to the kernel.

An overview of the *ELF* is given on the *Wiki* though the student may want to read a more detailed documentation.

`kernel_preload`

This function loads into memory the number of sectors given into `%al`, from the drive specified in `%dl`, starting from the second sector. Note that it is the responsability of this function to initialize the drive.

In addition, the function must switch to the *Protected Mode* and return into `%eax` the address of the memory area where sectors have been loaded *i.e.* the location of the *ELF* in memory. Since the code calling this function is obviously running in 16-bit, an additional 16-bit zero will be pushed before calling `kernel_preload` in order to successfully retrieve the return address in its 32-bit format.

Noteworthy is that this function must succeed even if the kernel image spans over multiple sectors.

`elf_load`

This function parses the *ELF* file located at the address given in `%esi`, loads its segments into memory, and executes its entry point, assuming that the *kernel_preload()* function has been called prior to this function.

Chapter 6

k1

k1 is the first stage taking place within the *kaneton* microkernel. This stage focuses on the event processing mechanism, one of the most fundamental kernel component. This mechanism provides the kernel the capacity to react to internal or external stimuli and to perform computations based on their nature.

The event processing mechanism enables the kernel to catch errors, known as *exceptions* on *Intel* architectures, such as division by zero, page faults *etc.* but also to communicate with external devices. Indeed, although the easiest way to communicate with devices consists in probing *I/O* ports until the device's state changes to something expected, *interrupt*s provide a mechanism consisting for devices to generate asynchronously an event which will, as its name suggests, suspend the kernel's activity to handle the interrupt.

Such events are extremely important as they are also used to simulate multitasking *i.e.* the ability to execute multiple threads of execution in "parallel". Note that the mechanism underlying this concept will be discussed in *k3*.

In this stage, students will therefore address the following concepts:

- **Event Mechanism Theory**

  The way event processing mechanisms are implemented depending on the architecture will be studied through the lectures.

- **IDT - Interrupt Descriptor Table**

  The *IDT* will be manually build and managed in order to specify the *CPU* how to handle the interrupts being exceptions, *IRQ - Interrupt Request*s *etc.*

- **ISR - Interrupt Service Routine**

  Once an interrupt is triggered, a handler—referred to as an *ISR* on *Intel* architectures—is launched. This handler is responsible for saving and restoring the state of the interrupted execution context among others.

- **PIC - Programmable Interrupt Controller**

  The student will also discover the existence of a chipset known as the *PIC* whose purpose is directly linked to the interrupt handling.

6.1 Submission

Unlike *k0*, students are not supposed to provide a specifically built snapshot. Instead, the test system will archive the student's code in a proper way and submit it.

For the process to work, students must first take care to remove the *k0* snapshot which may be lying at the following location: `test/client/snapshot.tar.bz2`. If this snapshot is not removed, the test system will use it for the submission process which will result in *k0*'s code being submitted for the *k1* stage.

Besides, students should switch their capability so as to use the one provided for the stage *k1* and above.

6.2 Requirements

Every student should read the *IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1* which is available on the *Wiki*. More precisely students should focus on the chapter *Interrupt and Exception Handling* which contains all the necessary material to complete this stage.

Finally, if not already the case, students should also read the chapters *Protected-Mode Memory Management* and *Protection* as interrupt handling makes use of the protected mode capabilities.

6.3 Assignments

The *k1* stage, focusing on the event processing mechanism, takes place entirely in the *event* manager. However, since every processing is highly machine-dependent, related code can also be found in the *glue*, component whose main task is to redirect calls to the functionalities provided by the *platform* and *architecture*.

Although, as students will soon notice, the provided snapshot includes a complete event manager's core, the glue is completely empty while the architecture does not provide any functionality related to the event processing mechanism.

Therefore, the *k1* assignments consist for the student to equip the kaneton educational implementation with the necessary functionalities making the event manager fully operational.

Though this stage tries to give as much freedom as possible to students regarding their implementation, skeleton source files have been provided, hence acting as entry point for students to start experimenting.

6.3.1 Core

Although the event manager is obviously the most important manager when it comes to the event processing mechanism, the student should consider the timer manager as well.

Indeed, the timer manager relies on the underlying hardware timer event for providing the possibility to trigger functions after some time. The student may be interested in this manager as being a perfect application of the event manager.

event

- `kaneton/core/event/event.c`
- `kaneton/core/include/event.h`

The event manager's core provides the high level interface enabling the kernel and its servers to plug an event handler with a specific event number *i.e.* event identifier. Note that event identifiers are machine-dependent; for instance the event `#32` on *Intel* architectures represents the timer. The event manager's core thus does not understand the meaning of the event identifiers and it is therefore the responsability of the machine, through the glue component, to perform the necessary operations according to the underlying platform/architecture.

Although the manager's core is complete, the student will probably find it useful to browse its source code to better understand its purpose.

6.3.2 Glue

As mentioned earlier, the timer manager relies on the hardware timer event. The student will notice in `kaneton/machine/glue/ibm-pc.ia32/educational/timer.c` that this event is actually reserved and linked with a handler located within the timer manager's core.

event

- `kaneton/machine/glue/ibm-pc.ia32/educational/event.c`
- `kaneton/machine/glue/ibm-pc.ia32/educational/include/event.h`

The event manager's glue makes the binding between a core request and the underlying platform and architecture functionalities.

The student will notice that the glue is empty when it comes to the event manager. Indeed, it is the student's responsability to complete the glue in order to equip some of the event manager's functionalities with a machine-specific behaviour.

One can notice an empty array referred to as `glue_event_dispatcher`. The role of such so called dispatchers is to receive the requests from the core, *i.e.* whenever the core calls `machine_call()`, and handles them by taking into account both the platform and the microprocessor architecture, *IBM PC* and *IA-32* for instance.

The dispatcher must be filled with the symbols of the functions the glue wishes to support. The dispatcher composition is located in the core header of the manager. For instance, considering the event manager, one can retrieve the event dispatcher composition `d_event` in the header file `kaneton/core/include/event.h`. For instance, should one want to catch the calls to `event_reserve()` in the glue, a function would have to be written, say `glue_event_reserve()`, and registered in the `glue_event_dispatcher` array at the sixth position.

6.3.3 Architecture

The architecture must be completed in order to provide functionalities for controlling the microprocessor's capability to handle the interrupts and exceptions.

For that purpose, skeleton source files are provided for the student to implement functions regarding the management of the *IDT - Interrupt Descriptor Table* and the set up of the interrupt handlers.

idt

- `kaneton/machine/architecture/ia32/educational/idt.c`
- `kaneton/machine/architecture/ia32/educational/include/idt.h`

The file `idt.c` must be completed by students in order to implement functions related to the *IDT* management such as building an *IDT* from a given address, inserting an entry in the *IDT* according to some arguments, deleting an existing *IDT* entry *etc.*

Note that in order to create *IDT* entries, students should have a look at the function `architecture_gdt_selector()` which enables the caller to generate a segment selector according to several parameters.

handler

- `kaneton/machine/architecture/ia32/educational/handler.c`
- `kaneton/machine/architecture/ia32/educational/include/handler.h`

Intel interrupts can be classified according to their purpose into one of the following categories: exceptions, *IRQ - Interrupt Request*s, *IPI - Inter-Processor Interrupt*s and system calls *a.k.a. syscall*s. Noteworthy is that *IPI*s can be ignored since the educational implementation only supports mono-processor architectures.

The objective is for students to implement the low-level interrupt handlers which are triggered whenever the associated interrupt is received. Such a handler, sometimes referred to as *lower half*, performs two basic operations. First, the environment is secured for the execution of the interrupted thread to be resumed properly, once the interrupt treated. Second, the high-level event handler function registered through `event_reserve()` is triggered; this function is sometimes referred to as the *higher half*.

The file `handler.c` should thus contain the interrupt handlers associated with all the interrupts the kernel wishes to handle. Besides, this file should make use of the functions provided in `idt.c` for setting up the system's *IDT*.

architecture

- `kaneton/machine/architecture/ia32/educational/architecture.c`
- `kaneton/machine/architecture/ia32/educational/include/architecture.h`

The student should also take care to record, whenever appropriate, the error code associated with the interrupt in `_architecture.error`.

A special attention must be given to the `_architecture` structure. Indeed, because this structure is being used by the testing system, its organisation should never be modified.

6.3.4 Platform

PIC

- `kaneton/machine/platform/ibm-pc/pic.c`
- `kaneton/machine/platform/ibm-pc/include/pic.h`

The platform plays a minor role in the event processing mechanism. However, students should have a look at the *PIC - Programmable Interrupt Controller* which may prove useful in this stage.

6.4 Example

This section presents an example which students can use to better understand the use of the event manager. This example illustrates a breakpoint exception being generated manually through the `int` assembly instruction. The exception is then caught by the kernel which notices that an event handler has been registered and thus triggers it.

```
void                    exception_bp(t_id                       id,
                                     t_vaddr                    data)
{
  printf("[event %qd] breakpoint exception caught\n",
         id);
}

void                    example(void)
{
  if (event_reserve(ARCHITECTURE_IDT_EXCEPTION_BP,
                    EVENT_TYPE_FUNCTION,
                    EVENT_ROUTINE(exception_bp),
                    EVENT_DATA(NULL)) != STATUS_OK)
    {
      printf("[event_reserve] error");
      return;
    }

  asm volatile("int $3");

  if (event_release(ARCHITECTURE_IDT_EXCEPTION_BP) != STATUS_OK)
    {
      printf("[event_release] error");
      return;
    }
}
```

Note however that this example is specific to exceptions. In order to test *IRQ*s, students should first activate the event system by invoking the `event_enable()` function.

6.5 Advices

This section contains advices that students are welcome to consider:

- Students should write macro-functions for setting the fields within the *IDT* entries, making it easier to transform a single address into scattered fragments placed at different position within an entry.

  One can take the example of *GDT*-specific macro-functions which are located in `kaneton/machine/architecture/ia32/educational/include/gdt.h`

  In addition, the opposite should also be set up in order to easily retrieve a field from an *IDT* entry.

- The interrupt handlers located in `handler.c` should be composed of two parts.

  The first one, composing the entry point of the *ISR - Interrupt Service Routine*, should be written in assembly in order to prevent the compiler from generating additional instructions.

  The second one, called by the first one and written in *C*, triggers the high-level event handlers registered for this event.

- The assembly part of the interrupt handlers are actually all very similar, differing slightly depending on the nature of the interrupt: exception, *IRQ*, *IPI etc.* and depending on the presence of an error code.

  Therefore, instead of writing all these assembly functions manually, one could rely on a macro-function for generating the assembly code automatically.

- In order to ease the debugging process, students should get used to writing functions for dumping the state of the processor's structure, such as the *IDT*.

- An interrupt handler, noticing that no high-level handler has been registered for this event identifier, should display a message on the console warning that this interrupt was unexpected.

  This way, the developer will easily notice exceptions occuring as well as unhandled IRQs.

- The `event_enable()` function must behave in a particular way for the test system to work properly. In addition to activating the interrupt mechanism, this function must also trigger an event right away.

  Indeed, most tests call `event_enable()` followed by a call to `event_disable()`, assuming that at least one timer interrupt will have been triggered in between for the actual test to be launched.

  Thus, if it happens that no timer interrupt is generated between the two calls, the test would actually fail. Students are therefore advised to force the generation of such an event.

- Students must understand the differences between exceptions and interrupts, especially in the context of kaneton.

  For instance, although exceptions can be triggered at any time, interrupts require the event mechanism to be enabled, through `event_enable()`, not to mention that interrupts must not be masked through the *PIC*.

Chapter 7

k2

The *k2* stage consists for students to provide the microkernel a complete set of memory management functionalities.

Modern kernels rely on virtual memory for ensuring isolation between the various entities, thus creating what is commonly referred as *address spaces*.

Virtual memory offers many advantages compared to the ancient segmentation model present on *Intel* architectures. Besides, though implemented differently, most architectures now provide a paging system, hence easing the task consisting in porting a kernel on several microprocessor architectures.

Throughout this stage, students will learn the notions listed below:

- **Paging**

  Students will have to implement low-level functions for setting up, updating and deleting *PD - Page Directories* and *PT - Page Table*s. In addition, students will have to implement higher-level functions in order to ease the common operation consisting in mapping a *region* to a *segment*.

- **Debugging**

  Unfortunately, the paging mechanism has the tendency to generate an awful number of subtile mistakes which are difficult to spot. Debugging therefore plays an important role in this stage as students able to quickly identify the source of the error will be far more efficient than the ones who cannot.

  Thus, students will have to rely on exceptions and the interrupt handling mechanism set up throughout *k1* in order to easily locate the page which is responsible for the error.

- **Hardware Techniques**

  Finally, students will learn that a number of hardware techniques and optimisations have been invented over the years such as the *Identiy Mapping* or the *Mirroring Technique*, some of which students may need to use.

7.1 Requirements

Every student should read the chapter titled *Paging* from the *IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*; this chapter contains everything necessary for implementing this stage.

In addition, students are encouraged to read the chapter *Protection* which contains material regarding the *Intel* ring-based protection mechanism.

7.2 Assignments

Although kaneton already embeds several managers related to memory management, the microkernel is devoid of an implementation on the *IA32* educational architecture.

The assignments for this stage consists for the student to implement the missing functionalities especially related to the *Intel* microprocessor architecture.

7.2.1 Core

The core is composed of four managers involved in the memory management.

The *segment* manager provides functions for managing the physical memory. On most platforms, including the *ibm-pc*, the physical memory is represented by the *RAM - Random Access Memory*, though not directly accessible. Thus, whenever the kernel or one of its servers needs memory to store information, the segment manager is requested.

The *region* manager provides the virtual memory capability enabling the kernel and its servers to access segments directly.

The *as* manager provides the possibility to reserve an address space, itself containing a set of segments as well as a set of regions mapping some of the segments.

Finally, the *map* manager provides high-level functions for easily reserving, resizing and releasing memory without dealing with segments and regions or any identifier.

as

- `kaneton/core/as/as.c`
- `kaneton/core/include/as.h`

Every address space is linked to its task and contains a set of segments and a set of regions. In other words, an address space object references all the physical memory owned by the address space along with the areas of virtual memory mapping the segments, making these segments accessible directly.

Although the as manager's core is complete, students should have a look at it.

region

- `kaneton/core/region/region.c`
- `kaneton/core/include/region.h`

The region manager provides a collection of functions for managing regions *i.e.* areas of virtual memory.

As for the as manager, this manager is already complete but students should browse through its source code to fully understand its purpose.

7.2.2 Glue

The segment manager's glue implements the segment manager's machine through the *Intel* segmentation model while the region manager's glue relies on the paging mechanism for providing virtual memory.

The particularity of the *ia32* architecture compared to the kaneton design is that *Intel*'s paging system assigns permissions to virtual pages while kaneton assigns permissions to segments *i.e.* physical memory. This difference requires the kaneton machine to adapt by setting pages' permissions according to the permissions associated with the segment to map.

as

- `kaneton/machine/glue/ibm-pc.ia32/educational/as.c`
- `kaneton/machine/glue/ibm-pc.ia32/educational/include/as.h`

The as manager's glue is already complete though the student may want to extend it.

A particular attention should be paid to the `glue_as_reserve()` function which initializes the address space depending on its nature, either the kernel address space, or one of the server's.

region

- `kaneton/machine/glue/ibm-pc.ia32/educational/region.c`
- `kaneton/machine/glue/ibm-pc.ia32/educational/include/region.h`

The region manager's glue must perform the necessary operations in order to satisfy the core request according to the underlying machine.

For instance, given a `region_reserve()` request, the glue should probably rely on the architecture in order to map the segment by updating the page directory and tables accordingly.

The student must therefore complete the region manager's glue by implementing the necessary functions in order to provide the virtual memory mechanism.

7.2.3 Architecture

The *ia32/educational* implementation provides mostly empty files with the exception of some skeleton functions in `paging.c`. Students are welcome to change everything as long as the name and prototype of the functions `architecture_paging_pdbr()`, `architecture_paging_map()` and `architecture_paging_unmap()` remain unchanged as these functions are directly referenced by the tests.

pd

- `kaneton/machine/architecture/ia32/educational/pd.c`
- `kaneton/machine/architecture/ia32/educational/include/pd.h`

This file should contain functions for manipulating *PD - Page Directories* including building a page directory inside a given memory location, inserting a page directory entry, deleting a page directory entry, mapping and unmapping a page directory *etc.*

Students should be careful when implementing such functions especially when it comes to mapping and unmapping a page directory. Students must keep in mind that the memory management functionalities provided by the core cannot be used for setting up a data structure in main memory because such functions will probably end up using the page-directory-specific functions located in this file. Students must therefore carefully choose the core functions to use when memory is needed in order to avoid infinite loops.

pt

- `kaneton/machine/architecture/ia32/educational/pt.c`
- `kaneton/machine/architecture/ia32/educational/include/pt.h`

The file `pt.c` is analoguous to `pd.c` but instead operates on *PT - Page Table*s.

paging

- `kaneton/machine/architecture/ia32/educational/paging.c`
- `kaneton/machine/architecture/ia32/educational/include/paging.h`

The `paging.c` file should contain functions for setting up the paging mechanism or making a given page directory the system's current one.

Besides, the student will have to complete the two most important paging functions: `architecture_paging_map()` takes a segment to map and updates the necessary page directory and tables accordingly while `architecture_paging_unmap()` does the opposite. Note that both functionalities should rely on the functions provided in `pd.c` and `pt.c` especially in order to temporarily map the necessary page tables and update them.

environment

- `kaneton/machine/architecture/ia32/educational/environment.c`
- `kaneton/machine/architecture/ia32/educational/include/environment.h`

Finally, the `environment.c` file should be updated. Indeed, five `FIXME` can be found in the source code, all related to memory management.

Note that the page directory set up by the boot loader can be retrieved in `_init->machine.pd`. Indeed, it is interesting to discover that the kernel, when launched, already evolves in a virtual environment.

The student can actually notice that the boot loader's page directory is imported in `architecture_environment_kernel()` and must then be cleaned from all the needless page table entries.

7.3 Example

The following example illustrates the as, segment and region managers by setting up a segment being shared between the kernel address space and another one.

```
extern m_kernel         _kernel;

t_status                 example(void)
{
  i_task                task;
  i_as                  as;
  i_segment             segment;
  i_region              region;
  o_region*             o;
  t_uint32*             data;

  /*
   * kernel
   */

  if (segment_reserve(_kernel.as,
                      ___kaneton$pagesz,
                      PERMISSION_READ,
                      SEGMENT_OPTION_NONE,
                      &segment) != STATUS_OK)
    CORE_ESCAPE("unable to reserve a segment");

  if (region_reserve(_kernel.as,
                     segment,
                     0x0,
                     REGION_OPTION_NONE,
                     0x0,
                     ___kaneton$pagesz,
                     &region) != STATUS_OK)
    CORE_ESCAPE("unable to reserve a region");

  if (region_get(_kernel.as, region, &o) != STATUS_OK)
    CORE_ESCAPE("unable to retrieve the region object");

  data = (t_uint32*)o->address;

  *data = 0x42;

  /*
```

```
   * server
   */

  if (task_reserve(TASK_CLASS_GUEST,
                   TASK_BEHAVIOUR_INTERACTIVE,
                   TASK_PRIORITY_INTERACTIVE,
                   &task) != STATUS_OK)
    CORE_ESCAPE("unable to reserve a guest task");

  if (as_reserve(task, &as) != STATUS_OK)
    CORE_ESCAPE("unable to reserve an address space");

  if (region_reserve(as,
                     segment,
                     0x0,
                     REGION_OPTION_NONE,
                     0x0,
                     ___kaneton$pagesz,
                     &region) != STATUS_OK)
    CORE_ESCAPE("unable to reserve a region");

  CORE_LEAVE();
}
```

7.4 Advices

This section contains advices that students are welcome to consider:

- Students should set a page fault handler displaying the instruction causing the fault along with the reason and the address of the page being accessed.

  Such a simple handler would provide students precious information for debugging this stage in which page faults are common.

- It is strongly advised for students to implement functions dumping the state of a given page directory or table.

  Once again, such functions would make the debugging process considerably simpler. Indeed, via these functions students could make sure the mappings are correct but also that some flags are not responsible for an unexpected behaviour.

- Note that permissions, supervisor or user, can be assigned to page directory/table entries. These permissions can actually be the cause of errors and should therefore be considered carefully.

- Finally, remember that any change to the current page directory/table hierarchy must be followed by cache invalidations in order to maintain consistency.

- Note that for some versions of *GCC*, bit-fields cannot be used when it comes to certain types such as `char`. As such, defining a structure as follows may result in a compilation warning and an incorrectly packed archive:

```
typedef struct
{
  t_uint32        addr0: 8;
  char            flags: 3;
  [...]
}                 __attribute__ ((packed)) t_pte;
```

It is therefore recommended to always rely on the kaneton types: `t_uint8`, `t_uint16`, `t_uint32`, `t_uint64` *etc.*

Chapter 8

k3

k3 consists in providing the kaneton microkernel with a multitasking environment.

kaneton relies on the concept of *task* as the execution entity. Every task is actually composed of an address space and one or more *thread*s. These threads are the actual active entity being managed by the *scheduler* which decides which one should be executed next. By switching between threads at a fast pace, the kernel gives the user the illusion to execute programs in parallel.

This stage will allow students to learn the following concepts:

- **Scheduler Algorithms**

  Throughout this stage's lectures, students will learn how scheduler algorithms can vary depending on the system's target, from general-purpose to specialized algorithms such as for real-time systems for instance.

  In addition, students will have to implement their own algorithm, hence providing the kaneton microkernel with a mechanism for controlling the threads' execution.

- **Hardware Context Switch**

  Students will also have to implement the low-level context switch mechanism which relies on interrupts. Through this implementation students will manipulate *TSS - Task State Segment*s and the *KIS - Kernel Interrupt Stack* among others.

8.1 Requirements

Every student will find the necessary material in the chapter *Task Management* of the *IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.*

In addition, students should not hesitate to dive into the chapters *Interrupt and Exception Handling* and *Protection* which contain material directly related to this stage.

8.2 Assignments

The objective of this stage is for students to *(i)* complete the scheduler manager's core by implementing their own algorithm and *(ii)* implement the hardware *Intel* context switch.

8.2.1 Core

The core contains four managers related to the execution. The *task* manager provides functions for managing the global execution entities *i.e.* tasks. The *thread* manager handle the active entities referred to as threads. The *cpu* manager controls the multiple processors present on the machine. Finally, the *scheduler* manager maintains the multiple schedulers, one for each *CPU*, each of which is responsible for the execution of a colletion of threads.

scheduler

- `kaneton/core/scheduler/scheduler.c`
- `kaneton/core/include/scheduler.h`

The student will find in `scheduler.h` the structure of the scheduler object `o_scheduler`. Let us recall that the scheduler manager is responsible for as many schedulers as the machine supports processors. Students are welcome to modify the structure of the scheduler object along with anything else as long as the scheduler manager's interface remains intact.

Note that although students are welcome to implement their own scheduler algorithm, the test system obviously expects the algorithm to be preemptive.

The following describe in detail every one of the functions which has been left to the student to implement.

`t_error` `scheduler_quantum(t_quantum quantum)`

This function modifies the scheduler manager's quantum *i.e.* the smallest unit of execution time; in other words threads are always executed for a duration multiple of the scheduler's quantum.

Note that in order to provide the best precision possible, the argument `quantum` should be a multiple of the timer delay.

This function's objectives are twofold. First the scheduler manager's quantum must be updated. Second, the timeslice—*i.e.* the duration a thread is allowed to be executed—of the threads registered in the scheduler, being active or not, should be re-computed since the quantum has changed.

`t_error` `scheduler_yield()`

This function is one of the most important in the scheduler manager. Whenever this function is called by the running thread, its execution must be stopped immediately hence inevitably leading to an election followed by a context switch.

Students are advised to thoroughly study the way the whole context switch mechanism works on the *ia32/educational* machine before implementing this functionality.

Noteworthy is that this function should also ensure that another thread is scheduled *i.e.* that the thread relinquishing the execution will not end up being re-elected right away. Indeed, the student can notice in the task and thread managers that this function is sometimes used to remove a thread definitely from the scheduler, for instance after its death. If the thread happens to be re-elected, its execution will continue while the kernel assumes it had been removed. Note however that the thread yielding its execution could very much be re-elected soon after, with a single thread being elected in between.

The student must therefore find a way to relinquish the execution right away while ensuring that the next elected thread is different.

`t_error` `scheduler_elect()`

This function is at the heart of the scheduler manager. The role of the `scheduler_elect()` function is to, as its name suggests, elect the next thread to execute.

This function is called on a regular basis, more precisely every quantum milliseconds. Thus, the currently being executed thread may not have exceeded its timeslice—*i.e.* allocated time of execution—in which case the thread could be re-elected. However, depending on the student's algorithm, a thread with a higher priority may need to be scheduled in which case the current thread could be put on hold for some time.

Noteworthy is that the current thread whose state is no longer `THREAD_START` or whose task's state is no longer `TASK_START` should be removed from the scheduler's queues. For more information on this specific case, please refer to `scheduler_remove()`.

In order to ensure that the *CPU* is always executing something, this function must make sure to always elect a thread. How to execute code when there is no thread left to execute is for students to figure out.

Finally, should the scheduler's state change to `SCHEDULER_STOP`, this function must elect the special kernel thread *i.e.* `_kernel.thread` in order to come back to the very first thread which started the scheduler. Note that this is extremely important as required by the test system to operate. Should this functionality be missing, all the tests would probably time out.

`t_error` `scheduler_add(i_thread id)`

This function adds the thread identified by `id` to the scheduler for execution.

`t_error` `scheduler_remove(i_thread id)`

This function removes the thread `id` from the scheduler.

Note that a special case must be made if the thread to remove is currently being executed; the thread's execution should be relinquished immediately. Note that this function is always called whenever the thread's state or its task's is being changed from `START` to something else. For more information, please refer to `thread_stop()`, `thread_block()`, `thread_exit()` or their task manager's counterparts.

`t_error` `scheduler_update(i_thread id)`

This function is called whenever the priority of the thread or of its task's has changed. This function aims at updating the scheduler priority of the given thread `id`.

Note that, depending on the student's algorithm, updating the thread's scheduler priority may require the thread to be moved to another queue, for instance. In addition, the thread's timeslice may also need to be re-computed.

Students should thoroughly test the scheduler implementation, making sure that threads are properly elected before moving to the hardware context switch.

8.2.2 Glue

Although, once again, the code of the *cpu*, *task*, and *thread* managers' glue is already provided, students are advised to take a look at them, especially the thread manager's glue which makes use of the `context_setup()`, `context_build()` and `context_destroy()` functions which will be discussed later.

scheduler

- `kaneton/machine/glue/ibm-pc.ia32/educational/scheduler.c`
- `kaneton/machine/glue/ibm-pc.ia32/educational/include/scheduler.h`

The scheduler manager's glue already contains a number of functions. Among the most useful ones, `glue_scheduler_start()` which reserves the timer used for triggering the `glue_scheduler_switch()` handler on a regular basis. Note that the `cpu` argument is ignored because the machine knows that multiprocessor architectures are not supported. Therefore, a single timer is reserved whatever the value of `cpu`. On the other hand, `glue_scheduler_stop()` relinquishes the execution in order for an election to be triggered. Thus, `scheduler_elect()` will notice that the scheduler's state has changed to `SCHEDULER_STOP` and the execution will be handed to the kernel thread, as expected.

The `glue_scheduler_switch()` is at the heart of this glue. This handler is triggered after quantum milliseconds and takes care to elect a new thread and call the `architecture_context_switch()` function in order to prepare the hardware context switch. Indeed, as discussed next, the `architecture_context_switch()` function does not perform the hardware context switch *per se*.

Noteworthy is that these functions assume that the currently running or being elected thread is referenced through the *CPU*'s scheduler at `scheduler->thread`.

Students will probably need to add functions to this glue in order to complete the scheduler's functionalities on this machine.

8.2.3 Architecture

The *ia32/educational* architecture relies on the interrupts in order to provide a context switch mechanism. This concept is based on the fact that whenever a thread is interrupted, its context is saved on a stack so that, when returning from the interrupt through the `iret` assembly instruction, the *CPU* retrieves the context from the stack, hence resuming the thread's execution. The context switch mechanism basically consists in *"replacing"* the context to be resumed in order for the processor to retrieve the context of another thread, hence switching the execution from the interrupted thread to another.

Note that the particularity of the *ia32/educational* architecture lies in the fact that a special stack, known as the *KIS - Kernel Interrupt Stack*, is used for handling interrupts. Indeed, while in the stage *k1* students were free to handle the interrupts their way, it is mandatory in *k3* to comply with this design.

Therefore, whenever an interrupt is triggered, the assembly part of the interrupt handler must save the necessary information regarding the interrupted thread. Then, the address space of the kernel must be loaded so as to set up the special interrupt-specific stack known as the *KIS*. Thus, the high-level interrupt handler is called and operates in the kernel address space, on a specific stack. Finally, once the interrupt handled, the kernel does the opposite actions by coming back from the *KIS* to the stack on which the interrupted thread's context has been saved, of another thread to resume. Then, the thread's context is restored so as for the *CPU* to resume its execution.

context

- `kaneton/machine/architecture/ia32/educational/context.c`
- `kaneton/machine/architecture/ia32/educational/include/context.h`

The *ia32* context management is composed of several functions which are briefly described next.

The `architecture_context_build()` function takes the identifier of a thread, and sets up its initial context depending on its privilege. Indeed, let us recall that while the context of an interrupted thread evolving in *ring0* is stored on its current stack, this is not the case for threads with lower privileges. Indeed, such threads possess an additional stack—known as the *pile* in kaneton and the *kernel stack* in many other kernels—on which their context is stored when interrupted. The function `architecture_context_destroy()` does the opposite by releasing the resources related to the thread's low-level context.

In addition, the functions `architecture_context_set()` and `architecture_context_get()` manipulate the given thread's context. Note that the context which these functions assumed to find on the thread's stack or pile is given by the structure `as_context`.

Finally, the `architecture_context_setup()` function initializes the context switch mechanism by reserving the *KIS - Kernel Interrupt Stack i.e.* the stack used specifically to handle interrupts. Moreover, this function allocates and initializes the system's *TSS - Task State Segment*. To better understand why a *TSS* is used, students are advised to read the chapters given in the *Requirements* section.

The objective of this stage is for students to implement the `architecture_context_switch()` function along with the macro-functions `ARCHITECTURE_CONTEXT_SAVE()` and `ARCHITECTURE_CONTEXT_RESTORE()`. Note however that students are welcome to modify the whole structure of the context managenent system or even re-implement everything they need.

Since the context switch mechanism relies on interrupts, students will have to modify their interrupt handling system in order to use both `ARCHITECTURE_CONTEXT_SAVE()` and `ARCHITECTURE_CONTEXT_RESTORE()`. Indeed, until now the only thread being executed, hence interrupted, was the kernel thread which evolves in *ring0*. The save/restore mechanism will therefore need to be adapted in order to support non-*ring0* threads.

The `architecture_context_switch()` function, which is called by `glue_scheduler_switch()`, should prepare the context switch by recording, in a structure such as `_architecture`, the information required for `ARCHITECTURE_CONTEXT_RESTORE()` to restore the context of the thread to resume. Then, when returning from the interrupt handler, the `ARCHITECTURE_CONTEXT_RESTORE()` macro-function will finally be called whose role will be to set up the information related to the thread to resume so that the *CPU* restore its registers.

environment

- `kaneton/machine/architecture/ia32/educational/environment.c`

The design of kaneton implies that servers address spaces only contains the minimal shared code and data mappings required to perform a context switch. In a nutshell, only the handler code and data sections lying within the kernel code segment should be mapped in a task.

The goal in this step is to implement the mapping of these sections of the kernel code segment when a server address space is initialized, in other words, in the `architecture_environment_server()` function. You will need to carefully read and understand the linker script, especially how to retrieve the bounds of each relevant section to perform its reservation in the task's address space.

handler

- `kaneton/machine/architecture/ia32/educational/include/handler.h`

The objective here is to make the necessary changes to your interrupt handlers in order to put them in the appropriate section. Again, you must look at the kaneton linker script and understand what it defines to do so.

architecture

- `kaneton/machine/architecture/ia32/educational/architecture.c`
- `kaneton/machine/architecture/ia32/educational/include/architecture.h`

The `_architecture` variable whose structure is given in `architecture.h` can act as a bridge between the `architecture_context_switch()` and the `ARCHITECTURE_CONTEXT_RESTORE()` macro-function.

The student will notice that this structure contains important information such as the kernel *PDBR - Page Directory Base Register*, the location of the *KIS - Kernel Interrupt Stack* among others.

8.3 Example

The following example illustrates how the scheduler manager can be used to execute threads. The reader may be surprised to see `event_enable()` immediately followed by `event_disable()`. However, one must understand that once the scheduler has been started and the events enabled, the thread will start being scheduled until one of those threads calls `scheduler_stop()`. Indeed, remember that the `scheduler_elect()` function, noticing the change of state to `SCHEDULER_STATE_STOP`, will elect the kernel thread for execution which happens to be the very first thread, the one which enabled the events. Therefore, resuming its execution, the kernel thread continues and disables the events right away.

```
extern m_kernel       _kernel;

void                  example_thread(void)
{
  i_cpu               cpu;

  if (cpu_current(&cpu) != STATUS_OK)
    {
      printf("unable to retrieve the current CPU\n");
      return;
    }

  if (scheduler_stop(cpu) != STATUS_OK)
    {
      printf("unable to stop the scheduler\n");
      return;
    }

  printf("unreachable");

  while (1)
    ;
}

t_status               example(void)
{
  i_thread            thread;
  i_cpu               cpu;

  if (thread_reserve(_kernel.task,
                     THREAD_PRIORITY,
                     THREAD_STACK_ADDRESS_NONE,
                     THREAD_STACK_SIZE_LOW,
                     (t_vaddr)example_thread,
                     &thread) != STATUS_OK)
    CORE_ESCAPE("unable to reserve a thread");

  if (thread_start(thread) != STATUS_OK)
    CORE_ESCAPE("unable to start the thread");

  if (cpu_current(&cpu) != STATUS_OK)
    CORE_ESCAPE("unable to retrieve the current CPU");

  if (scheduler_start(cpu) != STATUS_OK)
    CORE_ESCAPE("unable to start the scheduler");

  if (event_enable() != STATUS_OK)
    CORE_ESCAPE("unable to enable the events");

  if (event_disable() != STATUS_OK)
    CORE_ESCAPE("unable to disable the events");

  CORE_LEAVE();
```

```
}
```

8.4 Advices

This section contains advices that students are welcome to consider:

- Students should take care to update the thread's context location *i.e.* `thread->machine.context` whenever entering in the interrupt handler in order for the kernel to access the interrupted thread's context through the `architecture_context_set()` and `architecture_context_get()`.
- The *TSS - Task State Segment* should be updated appropriately according to the nature of the thread to be resumed.
- Students should remember to update the registers whenever switching from an environment to another such as from the thread's to the kernel's or the other way around.
- For the purpose of the `scheduler_yield()` and `scheduler_elect()` which must ensure that a thread is elected, even though there may be no thread left, students could very well introduce a special thread.

Chapter 9

Going Further

This chapter details what students who succeeded in implementing the *kaneton* educational project could do next.

9.1 Stop

Considering that a student has learnt what he wanted to, he could just stop here.

9.2 Implementation

Another possibility could be for a student to continue his kaneton implementation by providing higher level functionalities.

9.3 Research

A student could also join the kaneton research team and contribute to an exciting project by thinking differently from people who work on *Linux*, *BSD* and other *UNIX*-like projects.

Indeed, kaneton does not intend to be a *UNIX*-like but rather tries to innovate by considering systems in different ways.

9.4 Teaching

Teaching computer systems through kaneton in a school is another way of contributing.

Chapter 10

Licenses

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

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

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

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

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

The next sections will contain these licenses' descriptions.

10.1 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.

10.2 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.