The OSLib Manual

Luca Abeni Gerardo Lamastra

March 1, 2025

Abstract

This is the manual of OSLib, a collection of low level functions aimed to help programmers in developing system software, ranging from small programs for embedded systems to complex Operating System kernels. Using OSLib, the system programmer can focus on the software itself, without caring about the interaction with the hardware. In this sense, OSLib is similar to the Flux OS Toolkit.

From another point of view, OSLib provides an “easy” access to the hardware, without introducing any useless abstraction, and can be seen as a generic support layer for any operating system service. In this sense, it is similar to the MIT ExoKernel . Note that OSLib does not force to use any particular OS structure, but can support any conventional or innovative structure, such as the monolithic one, the microkernel-based one, or the vertical structured one.

Chapter 1

Introduction

The OSLib is a collection of routines and data structures developed to help system programmers in implementing an OS or an application that directly accesses the hardware. In this sense it is similar to the Windows NT Hardware Abstraction Layer (HAL), the μChoices NanoKernel, or the HARTIK Virtual Machine (VM) Layer. On the other hand, the aim of the OSLib is not to abstract the hardware resources (like the cited works do). In fact, hardware resources abstraction can result in poor efficiency and flexibility, as stated by Engler et others (ExoKernel). The OSLib code, instead of abstracting hardware, provides a secure and easy access to it hiding implementation details (for example, the PIC or PIT programming, or the CPU tables management) and leaving to the OS developers the hi-level and conceptual part of the work.

1.1 Library Structure

OSLib is composed of some libraries, that can be compiled using the standard GNU tools (gcc, GNU binutils and GNU make) either under MSDOS or Linux (DJGPP for DOS and gcc for Linux have been successfully tested; gcc for other Unix systems or Cygnus gcc for Windows have not been tested yet, but will probably work too). The resulting GNUCoff or ELF MultiBoot compliant images can be loaded using a custom provided DOS eXtender (X) or using the GNU Grand Unified Boot Loader (GRUB).

The code is organized in three parts:

- the hardware support library (`xlib`), used to access the PC hardware;
- a subset of the OS independent part of the standard C library (`libc1`);
- the Kernel support library (`kl`), that is the component to use for writing OS code.

The hardware support library contains the boot code for starting up the system when a MultiBoot compliant loader is used, the code to access hardware structures such as the GDT, IDT, the interrupt controller, the code to detect the CPU, and some data structures containing informations about the system.

The OS independent part of the C library provides all the functions from libc that can be implemented without invoking system calls (typically the string management functions, the memory copy/move/compare, the math functions and similar). An important exception to this rule is represented by the `cprintf` function: it is similar to the standard `printf` function (with the difference that `cprintf` directly writes to the screen), and, since it needs to access the video memory, it depends by the OS (in particular, `cprintf` depends on how the OS remaps the video memory). In any case, since the OS code needs to output some informations for debugging or other purposes, this function is provided by `libc1`. `libc1` does not provide any input function.

The Kernel support library provides:

- the code for interrupt/exception handling;
- the code for thread management (thread creation/deletion, context switch...);
- the code for address space management
- the code for time management

1.2 Compiling and Using

The OSLib code is distributed as source code in a ZIP or tarball archive. The tarball contains the source to be compiled in a Unix system (only `chr(10)` at the end of each line), while the ZIP archive can be decompressed in MSDOS (`chr(13)+chr(10)`) or Unix source using the -a option of UNZIP.

In order to decompress the source tree, use `tar -xvzf llxxx.tgz` or `unzip -La llxxx.zip`; this command will create the tree shown in Figure 1.1.

The `ll` directory contains the header files with the definitions of the OSLib structures and the prototypes for the OSLib calls. It is organized in two subdirectories: the `i386` directory, containing the include files for the hardware support library, and the `sys/ll` directory, containing the headers for the OS support library.

The `lib` directory is the place where all the libraries will be put once compiled. Depending on the distribution, the `lib` directory is in the archive or will be created at compilation time by the `make` command.

```
oslib--------ll--------i386
       |         |
       |         |
       |---lib   |----sys----ll
       |
       |---xlib
       |
       |---libc
       |
       |---libm
       |
       |---kl
       |
       |---examples
       |
       |---mk
```

Figure 1.1: The OSLib source tree.

The `xlib` directory contains the sources for the hardware support library; the `libm` directory is used to compile a modified version of the FreeBSD math library provided with OSLib; the `libc` directory contains the sources for the minimal C library, while the `kl` directory is the place where the Kernel support library source code resides.

The `examples` directory contains some examples showing how to use all the functionalities provided by OSLib.

The `mk` library contains some configuration files, used to compile the libraries in different host OSs: currently, the files to compile under MDSOS (using DJGPP V1 & V2) and the file to compile under Linux are provided. Moreover, a file to compile the H4 (S.Ha.R.K.) kernel is provided.

In order to compile the system, proceed as follows

- Configure the compiling system, installing the correct configuration file: copy the correct `mk/*.mk` file in `config.mk`
- make all the libraries, from directories `xlib`, `libc`, `libm`, and `kl`:

  ```
  cd xlib
  make install
  cd ..
  cd libc
  make install
  cd ..
  cd libm
  ...
  ```

- now the libraries are installed, and you are ready to use them. In order to test OSLib, you can compile the programs in the `examples` directory:

  ```
  cd examples
  make
  ```

A program compiled using the OSLib code can be run using the DOS eXtender, or using GRUB. In order to use the extender, boot MSDOS (or a 16 bit DOS compatible OS, such as FreeDOS), then copy `X.EXE` in the path, and finally use it: `X <program name>` (for example, try `X schedtest.xtn`). Once the program execution is terminated, it will nicely return to DOS.

In order to run a program through GRUB, put it in a GRUB accessible partition, then boot GRUB and enter the command prompt pressing `'c'`. Now, specify the compiled program as a kernel: assuming that you want to run `schedtest.xtn`, residing in the `/oslib/examples` directory on the first partition of your first hard drive, you have to type `kernel=(hd0,0)/oslib/examples/schedtest.xtn`. Finally, you can run the program typing `boot`. Once the program finishes, the system is halted and you must reboot it.

The OSLib code can be used including the adequate headers from the `include` directory, and linking the libraries from the `lib` directory. The compiler and linker options are set in the `config.mk` file: look at the makefile in the `examples` directory to see how to use it. The `examples` directory contains some simple programs to be browsed in order to learn how to use OSLib.

Chapter 2

The libraries

As said, the OSLib code and data structures are organized in various libraries, in order to increase the modularity and simplify the structure. A description of those libraries and of the header files that have to be included in order to use OSLib follows.

2.1 The header files

The `ll` directory contains the header files to be included for using OSLib. In particular, the directory structure tries to reflect the standard POSIX include directory. Hence, the include files of the minimal C library use the POSIX names and are distributed in `ll` and `ll/sys`.

The `xlib` headers are in the `ll/i386` directory; in particular, they are:

- `ll/i386/hw-data.h`: this header defines the basic data types and constants, and has to be included in order to use them
- `ll/i386/hw-instr.h`: this header defines the instruction needed to access hardware registers (or memory locations) as inlined functions
- `ll/i386/hw-func.h`: this header has to be included in order to use the functions that permits to directly access hardware, such as `halt()` and `reboot()`, `IDT_place()`, `GDT_place()` and `GDT_read()`, `addr2linear()`, and the `ll_context_*()` functions. Moreover, it contains the prototypes of `x_init()`, `x_end()`, `x_exc_bind()`, and `x_irq_bind()`
- `ll/i386/hw-arch.h`: this header has to be included for using the CPU/FPU detection functions `X86_get_CPU()`, `X86_get_FPU()`, `X86_is386()`, `X86_isCyrix()`, and `X86_hasCPUID()`
- `ll/i386/hw-io.h`: this header defines the I/O instructions as inline functions. It is included by `ll/i386/hw-instr.h` and must never be directly included by user programs

- `ll/i386/tss-ctx.h`: defines some macros for translating CONTEXTs in TSSs and vice-versa
- `ll/i386/sel.h`: defines some symbolic names for the predefined selectors used by the `xlib`
- `ll/i386/int.h`: defines some macros that simplify writing interrupt or exception handlers
- `ll/i386/pit.h`: this header has to be included in order to access the Programmable Interrupt Timer (PIT) through `pit_init()`, `pit_setconstant()`, and `pit_read()`
- `ll/i386/pic.h`: this header has to be included to access the Programmable Interrupt Controller (PIC) through `PIC_init()`, `PIC_end()`, `irq_mask()`, and `irq_unmask()`
- `ll/i386/linkage.h`: this header can be included by ASM files to generate correct C naming independently from the architecture/file format
- `ll/i386/defs.h`: this header defines some macros to be used to start/end header and C files
- `ll/i386/farptr.h`: this header contains the far pointer access from DJGPP
- `ll/i386/x-bios.h`: this header has to be included to call real mode functions through `X_callBIOS()`, or `vm86_init()` and `vm86_callBIOS()`, or to directly communicate with the eXtender through `x_bios_address()` and `X_meminfo()`
- `ll/i386/x-dosmem.h`: this header has to be included in order to manage real-mode (DOS) memory through `DOS_mem_init()`, `DOS_alloc()`, and `DOS_free()`
- `ll/i386/x-dos.h`: this header has to be included in order to access a FAT file system through dos (using the X-BIOS calls) with `DOS_fopen()`, `DOS_fclose()`, `DOS_fread()`, `DOS_fwrite()`, and `DOS_error()`
- `ll/i386/mb-hdr.h`: this header can be included by ASM files in order to easily generate a MultiBoot header
- `ll/i386/mb-info.h`: this header contains the definition of the MultiBoot Information (MBI) structure.

The `kl` headers are in the `ll/sys/ll` directory; in particular, they are:

- `ll/sys/ll/ll-func.h`: this header has to be included for using the `ll_context_create()`, `ll_context_setspace()`, and `ll_context_delete()`, `ll_init()`, `ll_end()`, and `ll_abort()` `ll_context_save()`, `ll_context_change()`, `ll_context_load()`, `ll_context_from()`, and `ll_context_to` functions

- `ll/sys/ll/event.h`: this header has to be included for using the event related functions, that are `event_init()`, `event_post()`, `event_delete()`, `irq_bind()`, and `ll_ActiveInt()`
- `ll/sys/ll/time.h`: this header has to be included for using the `gettime()` function. Moreover, it provides some macros for manipulating timespecs.
- `ll/sys/ll/event.h`: this header has to be included for using Address Spaces. In particular, it provides prototypes and data definitions for the `as_init()`, `as_create()`, and `as_bind()` functions.

2.2 The Hardware Support Library

The Hardware support library provides all the functions and data structures needed to access the hardware. In particular, it provides the code necessary to boot the OSLib application, to manage the CPU and the PC hardware, and to switch to Real Mode calling BIOS functions.

The booting code and data permits to create MultiBoot compliant executables and to interface the application with a MultiBoot compliant boot loader.

The CPU handling code and data permits to identify the CPU type and to manage the CPU tables (GDT and IDT), while the hardware managing code permits to access some specific PC hardware (PIT and PIC).

First of all, some basic data types are defined (in `ll/i386/hw-data.h`):

- `DWORD`: a 32 bit positive number;
- `WORD`: a 16 bit positive number;
- `BYTE`: an 8 bit positive number.

For each of these types, a `MAX_*` constant exists.

Based on the basic types, some important structures are defined:

- `struct gate`: the x86 gate structure;
- `struct descriptor`: an x86 segment descriptor;
- `union gdt_entry`: an x86 GDT entry: can be a gate or a descriptor;
- `struct tss`: an x86 Task descriptor.

All the constant that can be useful for initializing those structures are also defined. See Intel manuals for some explanations.

```
struct ll_cpuInfo {
                DWORD X86_cpu;
                DWORD X86_cpuIdFlag;
                DWORD X86_vendor_1;
                DWORD X86_vendor_2;
                DWORD X86_vendor_3;
                DWORD X86_signature;
                DWORD X86_IntelFeature_1;
                DWORD X86_IntelFeature_2;
                DWORD X86_StandardFeature;
}
```

Figure 2.1: The cpuInfo structure.

These are the functions provided by the hardware library to manage the CPU:

`void int X86_get_CPU(struct ll_cpuInfo *p)`

This function identifies the CPU present in the system and reports its characteristics in the `ll_cpuInfo` structure whose pointer is passed as an input. The `ll_cpuInfo` is described in Figure 2.2.

Basically, `X86_get_CPU()` checks if the CPU is a 386, a 486, or better; in the last case, it uses the `cpuid` instruction to obtain more information, otherwise it will use some custom code to determine the manufacturer. In particular, it is based on the following functions: `void int X86_is386(void)`
`void int X86_isCyrix(void)`
`void int X86_hasCPUID(void)`
It is recommended to invoke them through `X86_get_CPU()`, but sometime it can be useful to call one of the previous functions directly.

Similar code exists for detecting and initializing the FPU, but it still need some work.

Some other functions are provided to manage the most important CPU tables:

`void GDT_place(WORD sel, DWORD base, DWORD lim, BYTE acc, BYTE gran)`

This function permits to insert a new entry in the GDT; `sel` is the selector (equal to the entry number multiplied by *sizeof(struct gdt_entry)*), `base` and `lim` are the base and the limit of the segment described by the entry (and identified by `sel`), while `acc` and `gran` are the access and granularity bytes. They can be set using the constants defined in `ll/i386/hw-data.h`.

`DWORD GDT_read(WORD sel, DWORD *lim, BYTE *acc, BYTE *gran)`

This function permits to read an entry from the GDT, returning the base of the segment identified by the descriptor `sel`. Moreover, if `lim`, `acc`, and `gran` are not null, the limit, the access byte and the granularity of the segment are stored in `*lim`, `*acc`, and `*gran`.

`LIN_ADDR addr2linear(WORD selector, DWORD offset)`

This function can be used to translate an `<selector>:<offset>` address in a 32bit linear address. It uses `GDT_read` for obtaining the base of the segment identified by `selector`.

`void IDT_place(BYTE num,void (*handler)(void))`

This function permits to insert an entry in the IDT; `num` is the number of the interrupt, whereas `handler` is a pointer to the code that has to be specified as an handler for that interrupt.

After these very low level functionalities, the hardware support library provides some other facilities:

`void *x_init(void)`

This function initializes a minimal programming environment, defining a standard handler for all the CPU exceptions and hardware interrupts (through `IDT_place()`), identifying the CPU type, initializing the FPU, setting up a TSS for executing the code, and initializing the PIC.

Moreover, it returns a pointer to the MultiBoot Information structure (see Figure 2.2).

`x_exc_bind(int i, void (*f)(int n))` `x_irq_bind(int i, void (*f)(int n))`

These two functions permit to associate a handler to a CPU exception or to a hardware interrupt. Note that the handler is a C function, using the C conventions for passing the parameters. Since these two functions are based on the programming environment initialized by `x_init()`, they can be used only *after* that `x_init()` has been called.

`void x_end(void)` Restores the hardware settings to the DOS standards: it must be called on shutdown if the execution is expected to return to a 16bit OS.

Some other functions can be used for accessing the PIT:

`int pit_init(BYTE c, BYTE m, WORD tconst)`

This function initializes the PIT channel `c` (with `c = 0`, `1`, or `2`) in mode `m`, with the initial value of the counter equal to `tconst`. Returns a value `< 0` on error (if the requested channel does not exist).

int pit_setconstant(BYTE c, WORD val)

This function sets the PIT channel `c`'s counter to `val`. Returns a value `< 0` on error (if the requested channel does not exist).

`WORD pit_read(BYTE channel)`

This function reads the value of the counter of `channel`.

Other functions permit to call real mode interrupts, either returning to real mode or using the VM86 mode. Some of these functions are used to access a FAT file system using DOS (they works only if the application is invoked through the eXtender). See `ll/i386/x-*.h` for more details.

Some other functions directly remap the corresponding ASM instructions (these functions are implemented by the `xlib`):

- `cli()`
- `sti()`
- `halt()`
- `clts()`
- `BYTE inp(WORD port)`
- `WORD inw(WORD port)`
- `DWORD ind(WORD port)`
- `void outp(WORD port, BYTE data)`
- `void outw(WORD port, WORD data)`
- `void outd(WORD port, DWORD data)`

the following two functions can be used instead of `cli()` and `sti()`:

- `SYS_FLAGS ll_fsave(void)`: performs a `cli`, and return the previous value of the flags register;
- `void ll_frestore(SYS_FLAGS f)`: restores the flags register to `f`; can be used instead of a `sti()`.

Moreover, the library provides some inline functions for reading and writing some CPU registers' values:

- `get_CS()`
- `get_DS()`
- `get_FS()`
- `get_SP()`
- `get_BP()`
- `get_TR()`: returns the Task Register value
- `set_TR()`: sets the Task Register value
- `set_LDTR()`: sets the LDT address

2.3 The Kernel Support Library

The Kernel support library allows an OS developer to write interrupt handlers, binding them to hardware interrupts, to create threads and perform context switches, to crate address spaces and assign them to threads, and to manage the time.

Time management consists in reading time, and assigning execution time to threads through an event-based model. Hence, time management is performed using *events*: an event permits to execute some code (the event handler) at a specified time (the event rising time). When an event raises the event handler is called (with interrupt disabled).

Interrupts are managed in a similar way, allowing the programmer to specify the event handler for a special event that will raise when the hardware interrupt arrives.

Using the event mechanism it is easy to implement *temporal protection* (enforcing that a thread will never require too much execution time), while spatial protection is provided by OSLib through *Address Spaces*. An address space is a very basic abstraction encapsulating user data and code. Address Spaces are implemented using x86 segments: each Address Space is a different segment. If the user code uses only the default data and code selectors, the code running in an address space can not access other address spaces. As a default, OSLib provides a "flat" address space, mapping $1 \rightarrow 1$ all the physical memory.

Here is a list of the functions provided by `kl`:

`void *ll_init(void)`

This library function is used to initialize the Kernel Library: it detects the CPU, initializes the FPU, and sets the interrupt and exception handlers to default values.

As output, `ll_init` returns informations about the environment through a modified version of the MultiBoot Information (MBI) structure (the returned value is a pointer to such a structure). The MultiBoot Info structure is defined as shown in Figure 2.2.

Refer to the MultiBoot documentation for more informations about the fields behaviour. The only difference with the standard MultiBoot Info structure is that a new flag `MB_INFO_USEGDT` in the `flags` field is provided for informing that the program has been loaded through a DOS Extender, and two new fields are added. If the `MB_INFO_USEGDT` is set, the two new fields `mem_lowbase` and `mem_upbase` indicates the low memory and high memory starting addresses, otherwise the standard values (respectively $0x00$ and $0x100000$) have to be assumed.

```
CONTEXT ll_context_create(void (*entrypoint)(void *p), BYTE *stack, void *parm, void (*killer)(voi
WORD control)
```

This library function is used to create a new thread, allocating a CPU context for it. A thread is

```
struct multiboot_info {
/* MultiBoot info version number */
                      unsigned long flags;

/* Available memory from BIOS */
                      unsigned long mem_lower;
                      unsigned long mem_upper;
/* "root" partition */
                      unsigned long boot_device;

/* Kernel command line */
                      unsigned long cmdline;

/* Boot-Module list */
                      unsigned long mods_count;
                      unsigned long mods_addr;

                      union {
                            struct {
/* (a.out) Kernel symbol table info */
                                   unsigned long tabsize;
                                   unsigned long strsize;
                                   unsigned long addr;
                                   unsigned long pad;
                            } a;
                            struct {
/* (ELF) Kernel section header table */
                                   unsigned long num;
                                   unsigned long size;
                                   unsigned long addr;
                                   unsigned long shndx;
                            } e;
                      } syms;
/* Memory Mapping buffer */
                      unsigned long mmap_length;
                      unsigned long mmap_addr;
/*
Gerardo:  I need to add also the physical address base for
both low ( < 1MB) & upper ( > 1MB) memory, as X starts from DOS
which could have preallocated some of this memory...
For example, GRUB assumes that mem_lowbase = 0x0 &
mem_upbase = 0x100000
/
                      unsigned long mem_lowbase;
                      unsigned long mem_upbase;
};
```

Figure 2.2: The MultiBoot Information structure.

defined as an independent flow of execution and is characterized by the register set values, a private stack (used to initialize the `SS` and `ESP` registers), and an address space in which it executes. The code executed by a thread is defined by a function called *thread body* that takes as input a void pointer (passed at thread creation time).

The `entrypoint` parameter is a pointer to the thread body, the `stack` parameter is a pointer to a preallocated chunk of memory to be used as a stack for the new thread, while the `parm` parameter is a void pointer passed as parameter to the thread body when the thread is created. The `killer` parameter is a pointer to a function that will be called on thread correct termination (a thread terminates correctly when the execution arrives to the end of the body function). The `control` parameters defines some control flags associated to the thread.

The function allocates a free CPU context and initializes the register values using the passed parameters. The `EIP` register is initialized to `entrypoint`, and `ESP` is initialized to `stack`, whereas `DS`, `GS`, and `FS` are initialized to the default data segment and `CS` is initialized to the default code segment. As explained introducing Address Spaces, the default code and data segments remap one-to-one all the system memory ("flat" Address Space). All the other registers are initialized to standard values.

The return value is the identifier of the initialized CPU context.

`void ll_context_delete(CONTEXT c);`

This library function is used to free a CPU context when a thread is terminated. The `c` parameter is the identifier of the context to be freed. Note that the stack memory has to be explicitly freed, since `ll_context_delete()` does not free it.

`CONTEXT ll_context_save(void);`

This library function saves the CPU registers' values in the current CPU context and returns its identifier. In other words, the context associated to the thread executing when `ll_context_save()` is called is saved and its identifier is returned. It can be used to implement context switches in OS primitives, as shown in the following code:

```
SYSCALL(mysyscall(...))
{
                          CONTEXT oldContext, newContext;
                          ...

/* This must be the first primitive instruction */
                          oldContext = ll_context_save();
                          ...
                          OS primitive code

/* This must be the last primitive instruction */
                          ll_context_load(newContext);
};
```

Warning: if the virtual context switch mechanism is used, this function cannot be used (use `ll_context_from()` instead).

`void ll_context_load(CONTEXT c);`

This library call is used to load a new CPU context in the CPU, for performing context switches, as shown in the example above (see `ll_context_save()`). Note that `ll_context_load()` must be called **at the end** of a system call (immediately before re-enabling interrupts); if a system programmer needs to perform a context switch with interrupt disabled (in an event handler or in the middle of a system call), the *virtual context switch* mechanism have to be used. When virtual context switch is used, the context switch function only stores the new context ID in a temporary variable and performs the real context switch only when interrupts are enabled (see `ll_context_from()` and `ll_context_to()`).

`CONTEXT ll_context_from(void);`

This library function is similar to `ll_context_save()`, but can be called when the virtual context switch mechanism is used. In this case it returns the ID of the last context that have been selected to be loaded in the CPU.

`void ll_context_to(CONTEXT c);`

This library selects a thread to be dispatched: if interrupts are disabled and the context switch cannot be performed immediately, the real context switch will happen as soon as possible (when interrupts will be re-enabled). This is the `virtual context switch` mechanism.

`ll_context_to()` is similar to `ll_context_load()`, but uses the virtual context switch mechanism; if interrupts are enabled, they behave in the same manner.

`void ll_end(void);`

This function can be used in the shutdown code: if the application was started through the DOS Extender, `ll_end()` resets the PIT (and the rest of the hardware) to the standard DOS settings and prepares the system to return to MSDOS, otherwise it simply halts the system.

`void ll_abort(int code);`

This functions acts as safety place to go when any error occurs and the OS does not know which context is active. The function loads a safe context (with a safe stack), displays an error identified by the `code` parameter, and exits the OS support code (see also `ll_end`).

`void event_init(struct ll_initparms *l)`

This function sets the time management services up, by initializing the event queue and programming the Programmable Interval Timer (PIT) in a proper way. The PIT can be programmed in two different modes: the periodic mode and the one-shot mode. In periodic mode, the PIT is programmed to generate a timer interrupt each `tick` of T μseconds, specified by the user through the `l` parameter. In one shot mode, the PIT is dynamically programmed to generate an interrupt only when it is necessary to raise a programmed event. It is worth noting that the PIT mode only influences the error with which an event raises, but is invisible to the OS (the PIT mode can be changed simply changing the `event_init()` parameter, without any modify to the OS code).

The function takes as input a pointer `l` to a `ll_initparms` structure defined as follows:

```
struct ll_initparms {
                    DWORD mode;
                    TIME tick;
};
```

The `mode` field indicates the PIT mode (`LL_PERIODIC` or `LL_ONESHOT`), while the `tick` field indicates the tick size (for periodic mode only) in μseconds.

`TIME gettime(int mode, struct timespec *val)`

This function can be used to read the current system time. The system time can be read using different levels of precision (and different levels of overhead): currently only the `TIME_PTICK` and `TIME_EXACT` modes are implemented. The `TIME_PTICK` mode works only if the system timer is programmed in periodic mode, and returns the system time in ticks. It is characterized by a low overhead (small execution time). The `TIME_EXACT` mode reads the exact system time and returns it measured in μseconds.

The `mode` parameter can be `TIME_PTICK` or `TIME_EXACT` and specifies the time reading mode; the `val` parameter can point to a `timespec` structure that will be filled with the current time (if `val != NULL`) .

This function returns the read time in μseconds, or 0 if the reading fails.

`int event_post(struct timespec *time, void (*handler)(void *p), void *par)`

This function is used to create a new event, selecting an handler to be called at the specified time passing an user provided parameter to it. The `handler` parameter specifies the event handler (the function to be called when the event will raise), the `time` parameter indicates the time at which the event will raise, while `par` is a void pointer that will be passed as parameter to the event handler.

The function returns the identifier of the created event, or -1 if an error occurs (it can be due to the lack of free event descriptors, or to some other internal error). The event identifier is used to refer the event for modifying or deleting it (see `event_delete()`).

The OS support code programs the interrupt controller in a periodic or one-shot mode (see `event_init()`) so that an interrupt will be generated near to time `time` to call the event handler. The event handler is called as a response to the timer interrupt, with interrupts disabled, hence it **must** execute for not too much time (otherwise, interrupts will be left disabled for a long time). The timer mode can affect the error with which the event handler is called, but the code must be independent from the timer mode.

`int event_delete(int index)`

This library function is used to delete a posted event, identified by the `index` parameter. It returns 1 in case of success, -1 in case of failure.

`int irq_bind(int irq, void (*handler)(void *p), DWORD flags)`

This function can be used to associate an handler to an hardware interrupt; each interrupt is converted by the support code in an event, so the interrupt handler is identical to an event handler. The function checks if the requested interrupt is free, and in this case allocates it and assigns the handler to it. If the interrupt is already allocated (is not free), that is, a handler has been already associated to it, `irq_bind` returns an error and does nothing.

The `irq` parameter specifies the interrupt number, while `handler` is a pointer to the interrupt event handler, and the `flags` parameter defines some flags associated to the interrupt. In particular, the `FORCE` flag can be used to set a handler for an already allocated interrupt, and the `PREEMPTABLE` flag specifies that the handler can be called with interrupts enabled (interruptible handler).

FLAGS:

- the `PREEMPTABLE` flag permits to specify that the handler can be called with interrupts enabled (interruptible handler).
- the `FORCE` flag can be used to set a handler for an already allocated interrupt.

Interruptible handlers are useful to enhance system responsiveness, reducing the time in which interrupts are disabled and allowing to develop a preemptable OS. On the other hand, they must be used with caution, since mutual exclusion is not guaranteed in an interruptible handler.

The `FORCE` flag can be useful for removing an interrupt handler (use this flag with the `handler` parameter set to `NULL`.

`int ll_ActiveInt(void)`

This function returns the number of pending interrupts or event handlers.

`void as_init(void)`

This function initializes the Address Space management code. It must be called before using Address Spaces (`as_create()` or `as_bind()`).

`AS as_create(void)`

This library function can be used to create a new Address Space: it searches for a free Address Space descriptor and initializes it to an empty Address Space returning its identifier. The return value is the created Address Space ID in case of success, or 0 in case of failure.

`int as_bind(AS as, DWORD ph_addr, DWORD l_addr, DWORD size)`

This library function binds a chunk of physical memory to an Address Space. The `as` parameter is the Address Space identifier, `ph_addr` is the physical chunk start address, `l_addr` is the logical address in the `as` address space where the memory chunk has to be mapped, and `size` indicate the size of the memory chunk expressed in bytes.

Warning: currently, this function has been only partially implemented. In particular, since paging is not enabled, a single chunk of memory can be bound to an Address Space, starting from logical address 0.

`void ll_context_setspace(CONTEXT c, AS as)`

This library functions changes the Address Space in which a thread runs. Basically, ll_context_setspace() sets all the context `c` segment registers to the segment of the `as` address space. This function can be useful to create a new task:

1. Create a new context
2. Create a new Address Space
3. Set the context Address Space to the created one...

We need an example... Look at `examples/aspacedemo.c`.

2.4 Miscellaneous

Two functions `void *ll_alloc(DWORD size)` and `void ll_free(void *base, DWORD size)` are provided to allocate and free physical memory. They are provided only for convenience, but they should not be used: the memory allocator should be implemented in an upper level, using the informations returned by `ll_init()`.

Two functions `char *ll_context_sprintf(char *str, CONTEXT c)` and `void dump_TSS(WORD sel)` are provided for debugging purpose.