FreeLdr design and ideas (draft).

AUTHOR: Valery “valerius” Sedletski,

DATE 2007-12-11.

FreeLdr is a generic operating system loader, designed to be capable of loading DOS-like OSes (primarily, OS/2), as well as microkernel OSes (osFree, DROPS or any L4-based OS) and traditional monolithic OSes, like Linux. It shares many common features with GRUB, supports multiboot protocol and is partly based on GRUB sources. Though, it is greatly re-designed and includes many ideas from OS/2 boot process.

1. History of the project.

In 1999, David Zimmerli wrote an article in EDM/2 about a project to replace os2ldr (http://www.edm2.com/0705/freeldr/freeldr.html). He wrote a small rudimentary os2ldr-like program, which was supposed to begin a free os2ldr reimplementation from scratch. This program was started by the bootblock and tested the micro-FSD functions, opening a file and showing parameters, got from the bootblock. It was a stub for a loader. These sources were taken by osFree team and they were used as a starting point for osFree loader implementation. The osFree loader, like Zimmerli's program, is called FreeLdr. We took this name because it reflects the fact that our loader is free software and is a loader intended for use with osFree operating system. The first version was continued in 2003 by Yuri Prokushev and in 2005 by Sascha Schmidt. Then the development was paused. Sascha started to port portions of GRUB sources to incorporate multiboot protocol support, but did not finished the work. In the end of 2006, I took their results and finished the micro-FSD functions and combined them with completed working version of GRUB routines for multiboot loading. The loader worked. The L4 microkernel was loading using micro-FSD and started by kickstart bootstrapper. The loader worked in real mode and GRUB routines were ported to a realmode 16-bit environment (GRUB is mostly 32-bit).

The last time, the design was reworked and now I was redesigned the loader. It was decided to move most parts to 32-bits protected mode. Only small part continues to work in 16-bit real mode and is intended to be BIOS functions specific. This was made such a way because of the wish to make a possibility to easily port to other kinds of firmware, like EFI. So, the 16-bit portion must be kept as small as possible. Also, it was suggested that 32-bit code has less limitations and is easier to develop and extend (the extensions to the loader are easier to develop as in 32-bit code there is no 64 Kb segments and 640 Kb limit for available memory).

Now the new loader is partly working. I ported filesystem-specific parts of GRUB and made 32-bit micro-FSD's from them. The filesystem specific part is designed as a separate small (several kilobytes) module, and it is separated from generic part. Now isofs micro-FSD is starting os2ldr and os2ldr loads OS/2 kernel successfuly. But the missing part now is a mini-FSD.

When designing FreeLdr, there were the following design goals in mind, which determined the overall architecture of the loader:

1. Modularity. The loader must consist of several parts, which are replaceable by other parts, having the same interfaces with the rest of the loader.
2. To be OS-neutral. Possibility to load almost any OS. For that reason, FreeLdr employs a

multiboot protocol, the common protocol between OS loader and OS kernel, introduced in GRUB.
3. Compatibility with interfaces of traditional OS/2 loader, micro-FSD's.
4. Multiboot protocol compliance. (it is needed by L4, at least). But multiboot-specific part of the loader must be replaceable with standard OS/2 loader (os2ldr).
5. Like os2ldr, the FreeLdr must be independent from filesystem structure. This implies the use of BlackBox metaphor. The blackbox is something, doing its work internally and not exposing its internals to the outside. From the software point of view, the blackbox is a program, encapsulating its inner workings and exposing only its interfaces. The main kind of a blackbox is uFSD (micro-FSD, micro File System Driver), the small program, designed to abstract the loader from the filesystem structure. The loader only uses four uFSD functions to read files from the disk and details of internal filesystem structure are hidden from the loader.
6. And more. The FreeLdr goes even further, it introduces new kinds of blackboxes. There is a need to hide more details from the loader a) There are uXFD's (micro eXecutable Format Drivers), which abstract the loader from the structure of executable files; b) There are uDC's (micro De-Compressors) which are responsible for transparent uncompressing of files, read from the disk. They abstract the loader from the structure of compressed formats, like ZIP, GZIP, BZIP2 etc. GZIP uDC will be made also from GRUB sources (GRUB includes support for reading GZIP-compressed files and this will be transformed into modular design as a uDC) c) and also there are uPP's, micro Pre-Processors, responsible for transparently substituting variables in text files and for including one file into another (see below).
7. uFSD's and uXFD's are parts of IFS (installable file system) and IIF (installable image format) sybsystems, respectively. Both subsystems have a portion in kernel and in loader. Their goals are the support of arbitrary file systems or executable formats. In contrast, GRUB is designed to be monolithic (like UNIX kernel) and had no modularity at all. All filesystem-specific parts of the loader was statically linked in loader executable. The list of all filesystems was hardcoded, and to add a new filesystem, the loader must be recompiled. Also, file compression is hardcoded and executable formats support was also hardcoded. In FreeLdr, these parts are independent of the loader. To add a new uFSD, or uXFD, or uPP or uDC, no loader recompile is needed. The only needed thing is a new entry in pre-loader INI file, describing a new module.
8. The FreeLdr takes advantage of combined “loader and bootmanager” approach, it means that it allows to choose operating system to be loaded and in the same time it not only chooses a boot partition (the bootmanager role), but also loads files from disk (the loader role). These roles are combined, so when user chooses the boot menu item, it also chooses a set of boot parameters, which the loader then passes to the operating system. So, the loader is not only chooses OS to load and loads OS files from disk, it also allows the user to choose OS configuration parameters and passes them to the OS.
9. The bootmanager functionality of FreeLdr is designed to be replaceable with standard OS/2 bootmanager. So, if the user more likes the OS/2 bootmanager, it can switch off the loader bootmanager functionality (but then it loses the possibility to pass the OS boot parameters from the bootmanager menu) When the bootmanager functionality is switched off, the boot menu is simply disabled and the fixed boot script is executed.
10. The loader must have an abstraction layer hiding 16-bit BIOS calls and the firmware (BIOS or EFI) from the main part of the loader.

By 4-th and 9-th reasons, the decision was made that the loader must consist of, as minimum, two parts. The second part is interchangable, it may be os2ldr, or a multiboot loader (it is called stage1). The first part was called the pre-loader (It is called also stage0). It is started before stage1. It must implement the common feature set for os2ldr and multiboot loader.

The common features used by multiboot loader and os2ldr are a) uFSD, uXFD, uDC and uPP. It could

be useful if we could use them in old os2ldr too.

Also, the pre-loader serves as an abstraction layer between the machine firmware and stage1 of the loader.

The pre-loader in FreeLdr takes place of standard micro-FSD in OS/2 boot sequence.

2. Loader structure.

The loader consists of several parts, or modules. First, loader consists of loader itself, /boot/loader/freeldr module and a pre-loader. A pre-loader shields a main module from knowing different low-level details. It abstracts the loader from filesystem structure, the structure of executable modules and compression formats; from structure of tags in text files, read from the filesystem, and from details of accessing different kinds of terminals.

A pre-loader does its tasks by loading different blackboxes and using their functions. So, the pre-loader is a sort of blackbox manager.

3. Loader modules structure.

Each loader module (a main module, a pre-loader stages and blackboxes) have similar structure. The module is divided into two main parts. The main of them is a part containing 32-bit protected mode executable code. It goes after small 16-bit real/protected mode part. The 32-bit part can use call_rm() function to switch into real mode and execute a function in 16-bit part. For this, when module loads into memory, the module can be loaded at any address, including addresses above 1 Mb. Then 16-bit part is copied to buffer below 1 Mb, so, it can be executed in real mode.

16-bit module part starts with a sort of header. The header starts from 32-bit call instruction which jumps to init() function at the beginning of 32-bit part. After returning from init() function, RET instruction is executed and control is returned to the caller. So, the code looks like this:

```
        call init()
        ret
```

Note: These two 32-bit instructions are situated at the 16-bit code segment. It is done for the following purpose: The module is loaded first into the buffer and to call init() function, the starting module address is called. The call instruction at the module start just routes this call to correct init() function address.

After the two starting instructions, at the offset of 8h, goes the signature '$header$'. Then follows a load address of 16-bit part (below 1 Mb). It goes at the offset of 10h. Then, at the offset of 20h, the header ends.

4. Loader module initialization.

Each module is linked at some default base address. When it is loaded, it is first applied the fixups, according module .rel file. Then the 16-bit part is copied below 1 Mb at the address, specified in its header. After this, The init() function is called by calling the module starting address. The init()

function is given a pointer to some structure (in most cases, it is lip1 structure). This function main purpose is to set up fields of a given structure to point at module entry points. Such a way, other parts of loader can obtain information about module entry points. Also, init() function can do other initialization tasks, such as setting up GDT descriptors needed by this module. After calling the init() function the module is ready to use by other parts of the loader.

5. The pre-loader.

1) Blackboxes.

1. The uPP (micro preprocessor).
    - The idea of this feature is to have a small text file preprocessor (like C preprocessor, or similar, incorporated into freeldr_open() function (used for opening files by uFSD). This idea is inspired by Veit Kannegieser's program, called os2csm. Os2csm starts before os2ldr and presents a menu to user. User selects options from menu. Parameters, set by user are passed to OS/2 kernel and are substituted in config.sys like preprocessor defines. Os2csm hooks onto BIOS int 13h routines and when config.sys is read, itsubstitutes variables in it.
    - The uPP has similar idea. But unlike os2csm, it hooks onto freeldr_open() routine, not int 13h. This allows to preprocess not only config.sys, but any other text file.
    - The uPP is a common way to pass variables, set by user through menus, or ones, defined in a boot script, to OS programs through substituting these vars values in program command line or in its configuration files. It is very useful feature, you can see it in work in eComstation Demo CD, implemented in os2csm. We'd like to have such feature embedded in our loader as a standard feature.
2. The uDC (micro decompressor)
    - The GRUB feature to load files, compressed by GZIP, and decompress them while reading is also very useful. It is good for any kind of files. For some executable file formats, like LX, there is a built-in compression algorithms. For other formats, like ELF, there is not. So, to save disk space (especially on floppies), we could use this feature. But it would be more convenient to implement this feature as a blackbox, and do it modular, not to hardcode compression algorithm in loader, like in GRUB, but make it possible to easily add new compression algorithms without the need of recompile.
3. The uXFD (micro executable format driver)
    - Os2ldr contains routines for loading and relocating executable files of LX and NE formats. In GRUB, there is algorithms of loading of ELF and a.out formats. In both cases, these algorithms are hardcoded, and to add a new format, the loader must be recompiled. It would be very attractive feature, if we could do this modularily, without recompile. So, the decision suggest itself, to add executable format loading algorithms as a new kind of blackbox.
    - The uXFD has a one main function, a load() one, which takes a file image, loaded in memory, the relocation base address; loads and relocates it and returns ss:esp, cs:eip and other parameters. Also, it returns a result code.
4. The uFSD (micro file system driver)
    - The uFSD is the main kind of blackbox. I think, no need to explain, why it can be useful. We know its advantages from the OS/2 boot process.
5. The uT (micro terminal driver or terminal blackbox)
    - Its purpose is to abstract the loader from terminal access details. It provides a common interface to input/output symbols to different kinds of terminals, such as vga/ega console,

hercules console, serial console and others.
- All blackbox kinds allow us to avoid dependencies of the loader from different kinds of formats. The new formats can be added at any time without the need in recompile. In contrast, The GRUB has a monolithic design. The least change in loader needs a recompile. The GRUB is a good loader, it introduces almost universal protocol between loader and kernel and targeted primarily at microkernel systems (The GRUB was designed specifically for GNU HURD, not Linux, as it commonly considered). But although it is designed for modular microkernel system, it is monolithic by itself. And this is really bad :(. The NTLDR in Windows is also designed to be monolithic. The NTLDR (unlike the os2ldr) runs in protected mode. There are a realmode filesystem access functions in bootblock, by use of which the NTLDR is read from disk. But these functions (analogous to micro-FSD from OS/2) are not passed to NTLDR and are not used by NTLDR at all. Instead, there are a protected mode filesystem functions in NTLDR, by which the loader reads files, and these functions are hardcoded. This case is even worse than GRUB's. For a different filesystem, there must be a separate version of NTLDR – for fat32 the one,for NTFS the other. The NTLDR is tied to a filesystem type. In the case of FreeLdr, we have the choice in filesystem and even to add a new filesystem, no recompile is needed.

2) Interface between a pre-loader and a program which loads it (e.g., a bootsector)

Interface between a pre-loader and bootsector is fully compatible with the interface of OS2LDR and microfsd.

The interface is the following. The following info is passed in registers:

DL	–> boot disk BIOS number
DH	–> flags
DS:SI	–> boot disk BIOS parameter block (BPB)
ES:DI	–> FileTable structure

DH register bits:

0	- NOVOLIO	- minifsd does not use MFSH_DOVOLIO
1	- RIPL	- boot volume is not local (RIPL)
2	- MINIFSD	- minifsd present
3	- reserved	
4	- MICROFSD	- microfsd present
5-7	reserved and must be zero	

FileTable structure:

```
struct FileTable {
    unsigned short ft_cfiles; /* # of entries in this table            */
    unsigned short ft_ldrseg; /* paragraph # where OS2LDR is loaded     */
    unsigned long  ft_ldrlen; /* length of OS2LDR in bytes              */
    unsigned short ft_museg;  /* paragraph # where microFSD is loaded   */
    unsigned long  ft_mulen;  /* length of microFSD in bytes            */
    unsigned short ft_mfsseg; /* paragraph # where miniFSD is loaded    */
    unsigned long  ft_mfslen; /* length of miniFSD in bytes             */
```

```
    unsigned short ft_ripseg; /* paragraph # where RIPL data is loaded  */
    unsigned long  ft_riplen; /* length of RIPL data in bytes           */

    /* The next four elements are pointers to microFSD entry points     */

    unsigned short (far *ft_muOpen)
                     (char far *pName, unsigned long far *pulFileSize);
    unsigned long (far *ft_muRead)
                     (long loffseek, char far *pBuf, unsigned long cbBuf);
    unsigned long (far *ft_muClose)(void);
    unsigned long (far *ft_muTerminate)(void);
}
```

Also, there are flags in pre-loader file header <to be documented>, namely, microfsd size and a pre-loader size and also loader directory (where all files and configs are located). These must be set properly by caller.

There are three cases when loading a pre-loader:

1. DH <> 0 and valid microfsd and minifsd bit mask is set. – this means that the pre-loader is given control from standard 16-bit IBM microfsd. Then a pre- loader uses 16-bit microfsd for loading its own 32-bit microfsd's with subdirectories support.
2. DH = 0 and microfsd size field in pre-loader header is zero. – this means that a pre-loader and 32-bit microfsd is loaded to proper addresses and no relocation is needed. This is the case, for example, when loading with our FAT bootsector.
3. DH = 0 and microfsd size field in pre-loader header is <> 0. – this means that a pre-loader and 32-bit microfsd are loaded as a bundle (a pre-loader is concatenated with the microfsd – this is the case when loading from a bootblock) and they need to be relocated.

So, the pre-loader can be given 16-bit old-type microfsd as well as the new kind 32-bit microfsd. The 32-bit ones are generally used – they can be switched and different volumes can be mounted and subdirectories are supported. But 16-bit old ones also can be used. Their use is limited, though, as subdirectories are often not supported and volumes can't be switched. But it can be useful, if, 32-bit microfsd for some filesystem does not exist (as it is now for HPFS), or this can be useful when starting FreeLdr from OS2LDR by OS/4 team.