L4Ka::Kickstart

In our boot process we reuse L4Ka:Kickstart. L4Ka::Kickstart is a generic and extensible boot strapper for L4Ka::Pistachio, currently supporting the IA32 and AMD64 systems. L4Ka::Kickstart loads and configures the kernel with the architectural configuration parameters such as available and reserved memory areas. It further loads and registers the initial servers' memory location, and starts the kernel.

Note: Original kickstart from 0.4 L4Ka:Pistachio has broken Multiboot header. As result it is not works well. We use patched version of L4Ka:Kickstart.

Note: This document uses parts of Multiboot specification 0.6.95 and materials from L4Ka group.

KernelLoader/L4Ka:Kickstart interface

After L4Ka:Kickstart loaded and control passed to it machine must be in following state:

Machine state

Registers	Contains	Description
EAX	Magic value	Must contain the magic value 0x2BADB002; the presence of this value indicates to the operating system that it was loaded by a Multiboot-compliant kernel loader (e.g. as opposed to another type of kernel loader that the operating system can also be loaded from).
EBX	Pointer to Multiboot information structure	Must contain the 32-bit physical address of the Multiboot information structure provided by the kernel loader (see Boot information format).
CS	Must be a 32-bit read/execute code segment with an offset of 0 and a limit of 0xFFFFFFFF. The exact value is undefined.	
DS/ES/FS/GS/SS	Must be 32-bit read/write data segments with an offset of 0 and a limit of 0xFFFFFFFF. The exact values are all undefined.	
A20 gate	Must be enabled.	
CR0	Bit 31 (PG) must be cleared. Bit 0 (PE) must be set. Other bits are all undefined.	
EFLAGS	Bit 17 (VM) must be cleared. Bit 9 (IF) must be cleared. Other bits are all undefined.	

All other processor registers and flag bits are undefined. This includes, in particular:

- ESP The Kernel image must create its own stack as soon as it needs one.
- GDTR Even though the segment registers are set up as described above, the GDTR may be invalid, so the Kernel image must not load any segment registers (even just reloading the same values!) until it sets up its own GDT.
- IDTR The Kernel image must leave interrupts disabled until it sets up its own IDT.

However, besides this the machine state should be left by the kernel loader in normal working order, i.e. as initialized by the BIOS (or DOS, if that's what the kernel loader runs from). In other words, the operating system should be able to make BIOS calls and such after being loaded, as long as it does not overwrite the BIOS data structures before doing so. Also, the kernel loader must leave the PIC programmed with the normal BIOS/DOS values, even if it changed them during the switch to 32-bit

mode.

Boot information format

Upon entry to the operating system, the EBX register contains the physical address of a Multiboot information data structure, through which the kernel loader communicates vital information to the operating system. The operating system can use or ignore any parts of the structure as it chooses; all information passed by the kernel loader is advisory only.

The Multiboot information structure and its related substructures may be placed anywhere in memory by the kernel loader (with the exception of the memory reserved for the kernel and boot modules, of course). It is the operating system's responsibility to avoid overwriting this memory until it is done using it.

The format of the Multiboot information structure (as defined so far) follows:

Offset	Contains	Comments
0	flags	required
4	mem_lower	present if flags[0] is set
8	mem_upper	present if flags[0] is set
12	boot_device	present if flags[1] is set
16	cmdline	flags[2]
20	mods_count	present if flags[3] is set
24	mods_addr	present if flags[3] is set
28 - 40	syms	present if flags[4] or flags[5] is set
44	mmap_length	present if flags[6] is set
48	mmap_addr	present if flags[6] is set
52	drives_length	present if flags[7] is set
56	drives_addr	present if flags[7] is set
60	config_table	present if flags[8] is set
64	boot_loader_name	present if flags[9] is set
68	apm_table	present if flags[10] is set
72	vbe_control_info	must be filled by
76	vbe_mode_info	
80	vbe_mode	
82	vbe_interface_seg	
84	vbe_interface_off	
86	vbe_interface_len	

If bit 0 in the flags word is set, then the mem_* fields are valid. mem_lower and mem_upper indicate the amount of lower and upper memory, respectively, in kilobytes. Lower memory starts at address 0, and upper memory starts at address 1 megabyte. The maximum possible value for lower memory is 640 kilobytes. The value returned for upper memory is maximally the address of the first upper memory hole minus 1 megabyte. It is not guaranteed to be this value.

If bit 1 in the flags word is set, then the boot_device field is valid, and indicates which BIOS disk device the kernel loader loaded the Kernel image from. If the Kernel image was not loaded from a BIOS disk, then this field must not be present (bit 3 must be clear). The operating system may use

this field as a hint for determining its own root device, but is not required to. The boot_device field is laid out in four one-byte subfields as follows:

drive	part1	part2	part3

The first byte contains the BIOS drive number as understood by the BIOS INT 0×13 low-level disk interface: e.g. 0×00 for the first floppy disk or 0×80 for the first hard disk.

The three remaining bytes specify the boot partition. part1 specifies the top-level partition number, part2 specifies a sub-partition in the top-level partition, etc. Partition numbers always start from zero. Unused partition bytes must be set to 0xFF. For example, if the disk is partitioned using a simple one-level DOS partitioning scheme, then part1 contains the DOS partition number, and part2 and part3 are both 0xFF. As another example, if a disk is partitioned first into DOS partitions, and then one of those DOS partitions is subdivided into several BSD partitions using BSD's disklabel strategy, then part1 contains the DOS partition number, part2 contains the BSD sub-partition within that DOS partition, and part3 is 0xFF.

DOS extended partitions are indicated as partition numbers starting from 4 and increasing, rather than as nested sub-partitions, even though the underlying disk layout of extended partitions is hierarchical in nature. For example, if the kernel loader boots from the second extended partition on a disk partitioned in conventional DOS style, then part1 will be 5, and part2 and part3 will both be 0xFF.

If bit 2 of the flags longword is set, the cmdline field is valid, and contains the physical address of the command line to be passed to the kernel. The command line is a normal C-style zero-terminated string.

If bit 3 of the flags is set, then the mods fields indicate to the kernel what boot modules were loaded along with the kernel image, and where they can be found. mods_count contains the number of modules loaded; mods_addr contains the physical address of the first module structure. mods_count may be zero, indicating no boot modules were loaded, even if bit 1 of flags is set. Each module structure is formatted as follows:

0	mod_start
4	mod_end
8	string
12	reserved(0)

The first two fields contain the start and end addresses of the boot module itself. The string field provides an arbitrary string to be associated with that particular boot module; it is a zero-terminated ASCII string, just like the kernel command line. The string field may be 0 if there is no string associated with the module. Typically the string might be a command line (e.g. if the operating system treats boot modules as executable programs), or a pathname (e.g. if the operating system treats boot modules as files in a file system), but its exact use is specific to the operating system. The reserved field must be set to 0 by the kernel loader and ignored by the operating system.

Caution: Bits 4 & 5 are mutually exclusive.

If bit 4 in the flags word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

28	tabsize
32	strsize

36	addr
40	reserved (0)

These indicate where the symbol table from an a.out kernel image can be found. addr is the physical address of the size (4-byte unsigned long) of an array of a.out format nlist structures, followed immediately by the array itself, then the size (4-byte unsigned long) of a set of zero-terminated ASCII strings (plus sizeof(unsigned long) in this case), and finally the set of strings itself. tabsize is equal to its size parameter (found at the beginning of the symbol section), and strsize is equal to its size parameter (found at the beginning of the string section) of the following string table to which the symbol table refers. Note that tabsize may be 0, indicating no symbols, even if bit 4 in the flags word is set.

If bit 5 in the flags word is set, then the following fields in the Multiboot information structure starting at byte 28 are valid:

28	num
32	size
36	addr
40	shndx

These indicate where the section header table from an ELF kernel is, the size of each entry, number of entries, and the string table used as the index of names. They correspond to the shdr_* entries (shdr_num, etc.) in the Executable and Linkable Format (ELF) specification in the program header. All sections are loaded, and the physical address fields of the ELF section header then refer to where the sections are in memory (refer to the i386 ELF documentation for details as to how to read the section header(s)). Note that shdr_num may be 0, indicating no symbols, even if bit 5 in the flags word is set.

If bit 6 in the flags word is set, then the mmap_* fields are valid, and indicate the address and length of a buffer containing a memory map of the machine provided by the BIOS. mmap_addr is the address, and mmap_length is the total size of the buffer. The buffer consists of one or more of the following size/structure pairs (size is really used for skipping to the next pair):

1. 4 size

0	base_addr_low
4	base_addr_high
8	length_low
12	length_high
16	type

where size is the size of the associated structure in bytes, which can be greater than the minimum of 20 bytes. base_addr_low is the lower 32 bits of the starting address, and base_addr_high is the upper 32 bits, for a total of a 64-bit starting address. length_low is the lower 32 bits of the size of the memory region in bytes, and length_high is the upper 32 bits, for a total of a 64-bit length. type is the variety of address range represented, where a value of 1 indicates available RAM, and all other values currently indicated a reserved area.

The map provided is guaranteed to list all standard RAM that should be available for normal use.

If bit 7 in the flags is set, then the drives_* fields are valid, and indicate the address of the physical address of the first drive structure and the size of drive structures. drives_addr is the address, and

drives_length is the total size of drive structures. Note that drives_length may be zero. Each drive structure is formated as follows:

0	size
4	drive_number
5	drive_mode
6	drive_cylinders
8	drive_heads
9	drive_sectors
10-xx	drive_ports

The size field specifies the size of this structure. The size varies, depending on the number of ports. Note that the size may not be equal to (10 + 2 * the number of ports), because of an alignment.

The drive_number field contains the BIOS drive number. The drive_mode field represents the access mode used by the kernel loader. Currently, the following modes are defined:

0	CHS mode (traditional cylinder/head/sector addressing mode)
1	LBA mode (Logical Block Addressing mode)

The three fields, drive_cylinders, drive_heads and drive_sectors, indicate the geometry of the drive detected by the BIOS. drive_cylinders contains the number of the cylinders. drive_heads contains the number of the heads. drive_sectors contains the number of the sectors per track.

The drive_ports field contains the array of the I/O ports used for the drive in the BIOS code. The array consists of zero or more unsigned two-bytes integers, and is terminated with zero. Note that the array may contain any number of I/O ports that are not related to the drive actually (such as DMA controller's ports).

If bit 8 in the flags is set, then the config_table field is valid and indicates the address of the ROM configuration table returned by the GET CONFIGURATION BIOS call. If the BIOS call fails, then the size of the table must be zero.

If bit 9 in the flags is set, the boot_loader_name field is valid, and contains the physical address of the name of a kernel loader booting the kernel. The name is a normal C-style zero-terminated string.

If bit 10 in the flags is set, the apm_table field is valid, and contains the physical address of an APM table defined as below:

0	version
2	cseg
4	offset
8	cseg_16
10	dseg
12	flags
14	cseg_len
16	cseg_16_len
18	dseg_len

The fields version, cseg, offset, cseg_16, dseg, flags, cseg_len, cseg_16_len, dseg_len indicate the version number, the protected mode 32-bit code segment, the offset of the entry point, the protected

mode 16-bit code segment, the protected mode 16-bit data segment, the flags, the length of the protected mode 32-bit code segment, the length of the protected mode 16-bit code segment, and the length of the protected mode 16-bit data segment, respectively. Only the field offset is 4 bytes, and the others are 2 bytes. See Advanced Power Management (APM) BIOS Interface Specification, for more information.

The bit 11 in the flags must be zero.

L4Ka:Kickstart Command Line Options

L4Ka::Kickstart accepts a number of command line arguments that can be used to configure the system. The command line arguments are specified on the kickstart line in the boot loader's configuration file. Command line presented to L4Ka::Kickstart if flags[2] of multiboot information structure is set. Command line accesable via Cmdline pointer of Multiboot information structure. The following command line arguments are supported:

- maxmem=num[K|M|G]

Limits the amount of available physical memory in the system.

- kmem=num[K|M|G] (default 16M)

Reserves a specific amount of physical memory for kernel usage. The extra memory is recorded in the kernel interface page as a memory descriptor with a reserved memory type and picked up by the kernel during initialization.

- bootinfo=<on|off> (default on)

Enables/disables creation of a generic bootinfo structure. If enabled, the bootinfo location will be recorded in the bootinfo field of the kernel interface page.

- mbi=<on|off> (default on)

Enables/disables passing the multiboot info to the root tasks. If both bootinfo and mbi are enabled, the mbi location will be recorded in a special bootinfo record. If bootinfo is not specified, the mbi location is recorded in the bootinfo field of the kernel interface page.

- decode-all=<on|off> (default off)

Enables/disables decoding all ELF binaries found in the module list. The switch is useful for testing out multiple applications without having some special ELF decoder in the root task. The location and entry points for the binaries are recorded in SimpleExec bootinfo records. Kickstart will not complain if the decoded binaries overlap.

Memory Descriptors

L4Ka::Kickstart will, in addition to the regular memory descriptors defined in the L4 specification, insert a number of boot loader specific memory descriptors in the kernel interface page. These memory descriptors will have type 0xE (i.e., boot loader specific type) and one of the following subtypes:

- 0x0

Undefined/miscellaneous boot loader memory. Used for memory types that do not fit into any of the other categories.

- 0x1

Initial data structures required to boot strap the system (e.g., the multiboot info structure or generic bootinfo structure). These data structures may be safely freed up after the system has been initialized.

- 0x2

Initial L4 server (i.e., sigma0, sigma1, or root server).

- 0x3

Boot modules. Modules (simple files or executables) that are loaded by the boot loader.

From:
https://osfree.org/doku/ - **osFree wiki**

Permanent link:
https://osfree.org/doku/doku.php?id=en:docs:boot:kickstart

Last update: **2014/05/21 20:34**