Software &Systems Received 26.09.19
DOI: 10.15827/0236-235X.130.210-217 2020, vol. 33, no. 2, pp. 210–217

Trapper: an operating system bootstrapping package for IBM PC compatible computer systems

Y.I. Klimiankou [1], *Postgraduate Student, klimenkov@bsuir.by*

[1] *Belarusian State University of Informatics and Radioelectronics, Minsk, 220013, Belarus*

The paper presents an overview of the bootstrapping process on the IBM PC-compatible computer systems and proposes an architecture of the operating system bootstrapping package. The proposed package implements a framework for constructing boot images targeted at non-traditional operating systems like microkernel, an exokernel, unikernel, and multikernel. The bootstrapping package consists of three sets of independent boot modules and a Boot Image Builder application, which creates OS boot images. This application integrates and chains boot modules with one another to organize a complete bootloader chain. They are necessary to bring the operating system to a working state. The bootstrapping package architecture reflects the principal stages of the computer system boot process. Each set of boot modules is connected to the particular boot stage and forms a layer that is responsible for performing its own clearly defined set of functions and relies on clearly defined inter-layer interfaces to strictly isolate dependencies on the boot device, firmware and the specifics of the boot-loaded operating system.

The paper presents the implementation of the described architecture for boot image generation designed and implemented for a research multikernel operating system and explains how it boots up.

Additionally, the paper proposes the full separation idea of initialization code out of the operating system kernel and its movement into the independent OS loader module. Following this idea leads to the exclusion of the “dead” initialization-related program code from the OS kernel. In the commodity operating systems, such code runs only once during system boot, however, being the part of the kernel executable binary image, continues to occupy memory until the system shuts down.

Keywords: *boot image, loader, operating system, bootstrapping, image builder, IBM PC.*

IBM PC architecture was introduced in the early 80s and has rapidly become a de facto industry standard for a wide range of computer systems [1]. Modern IBM PC compatible computers use CISC processors with IA-32 and x86_64 instruction set architectures [2]. In this paper, we will mention only IA-32 while assuming them both, because x86_64 is a superset of IA-32.

IBM PC compatible computers are the example of ultimately complex and tightly integrated systems that consist of three major components: hardware, firmware, and software. Like any complex system, computers are characterized by inertia. Computers cannot perform user-defined work after power on immediately. Instead, they should pass through a comprehensive initialization process to come into a ready-for-use state. During this process, the computer system should properly initialize all its components. Furthermore, they should cooperate to accomplish this task. The main goal of software initialization is a bootstrapping of the operating system kernel.

This paper focuses on the Trapper architecture – the *operating system bootstrapping package* (OSBP) for IBM PC compatible computer systems. The bootstrapping package includes a three-layer set of independent modules and an application for their chaining during boot image assembling. We refer to this application as the *Boot Image Builder* (BIB).

A requirement to OSBP is to facilitate booting of various operating systems from a wide range of different boot devices and on machines with different types of firmware. Trapper design aims to be flexible and extensible to be able to support other types of boot devices, firmware, and a wide range of operating systems of different architectures with minimal additional investments.

Trapper design has two principal goals. First of all, it targets primarily operating systems with non-traditional architectures, while commonly used alternatives are focusing on operating systems with a monolithic kernel like Linux and Windows. Second, Trapper is a light-weight solution that creates a minimalistic boot image for explicitly specified boot scenario, while widespread solutions like GNU GRUB represent heavy-weight multi-purpose multiboot packages. Nowadays, the most experimental and research operating systems implement either their private booting environments or invest significant efforts into bidirectional adaptation between OS and existing booting solution.

Furthermore, we did not find any publications highlighting such issues. Almost all OS-related papers tend to omit this information. The secondary goal of this paper is to fulfill this gap. Moreover, Trapper supports an idea of complete separation of initialization code from OS kernel, which is especially advantageous for minimalistic OS kernels.

IBM PC boot process

A common way to initialize a computer system is the following. Immediately after power-up, each computer system processor performs hardware initialization and an optional *built-in self-test* (BIST). During this process, the system places each computer processor into a predefined well-known state. For example, IA-32 processors set registers to a known state, switch the CPU in real-address mode, invalidate the internal caches, *translation lookaside buffers* (TLBs), and the *branch target buffer* (BTB). Next, computer systems with multiple CPU cores on the board (multicore processor or multiple processors) execute protocol of multiple processor initialization. The goal of this protocol is to select one processor (*bootstrap processor* (BSP)) that will continue the boot process.

A computer places the rest of CPUs (*application processors* (APs)) in a halted state. The first instruction that is fetched and executed by BSP, following a hardware reset, is located at a hardwired physical address called a reset vector. For the IA-32 platform, the 16-bytes reset vector lies at the address FFFFFFF0H [3]. The EPROM containing the software-initialization code must be located at this address and performs a jump to the initialization code of computer system firmware that is also usually stored at ROM.

The IBM PC firmware, in its turn, performs several steps to prepare the system for work. The main action is RAM detection and initialization. First, firmware detects the installed RAM type and quantity and performs a simple memory test on it. Then, firmware detects, enumerates, configures, initializes and performs power-on self-test on every bus and almost every hardware device on the system. Firmware stores all information about the hardware configuration of a computer system as well as the memory map and then will pass it to the bootloader. After that, firmware chooses a boot device and reads a loader from it into the memory. Finally, it transfers control to the just read loader.

The main goal of the loader is to bring the entire boot image into the memory. For that purpose, the loader uses some configuration stored in non-volatile memory. This config contains information about the source and target location of the boot image, as well as its size. Types of the firmware and the used boot device define a loader code because different boot devices require different access methods. At the same time, the loader should be tiny and conform with the boot specification of the firmware.

The bootstrapping module depends on the firmware, and its goals are to bring the computer system into a predefined and firmware independent state and to collect and unify the information about a computer system. This information includes a multiprocessor configuration (number and IDs of CPUs present in the system) and a memory map and then is passed by the bootstrapper to the bootloader. The bootstrapper also switches the IA-32 processor from real into protected mode. The conceptual goal of the bootstrapper module is to encapsulate all firmware related activities and thus to isolate bootloader from firmware. The bootstrapper defines a firmware-independent interface to the bootloader to achieve this.

Finally, the bootloader finishes the operating system boot process. Bootloaders depend on an operating system. Their purpose is to prepare and initialize a kernel. A bootloader not only initializes kernel data structures but also switches a processor into the final mode of functioning (enables virtual memory and sets up descriptor tables), bootstraps application processors of the system, and initializes an initial set of applications. A bootloader terminates its job by passing control to the entry point of the first application that will already run under kernel control and continue the initialization of the OS environment and services.

Figure 1 shows the principal model of the computer system boot process. It demonstrates the main stages of the process and how the OSBP architecture maps them.

Green boxes in Figure 1 represent three layers of the OSBP architecture. They operate in the environment with one or two internal and one external interface. Both loader and bootstrapper are stick to the firmware interface while the bootloader sticks to the specification of the operating system and especially of its kernel. At the same time, a loader, a bootstrapper, and a bootloader are interfacing between each other via internal interfaces. The interface between a bootstrapper and a bootloader makes it possible to separate dependencies, while the interface between a loader and a bootstrapper creates an opportunity to generalize firmware dependencies. Trapper tries to use these opportunities and replicates the principal model

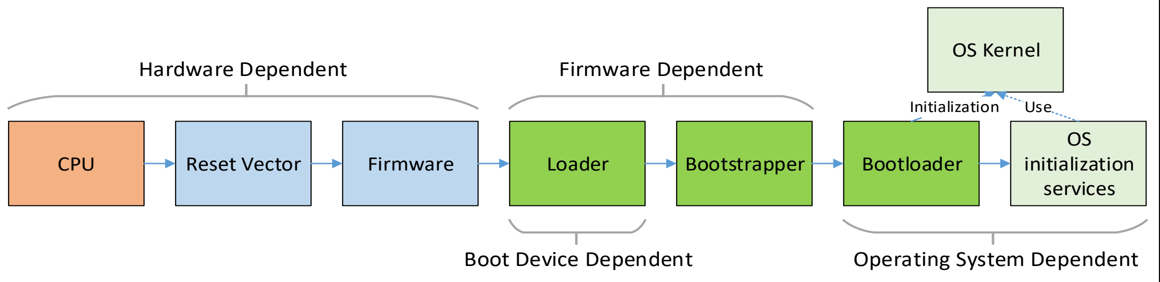

Fig. 1. A principal model of a computer system boot process

structure to create a highly flexible and extensible operating system bootstrapping package.

Trapper OSBP architecture

Figure 2 depicts the Trapper OSBPdesign, which allows flexible and easy construction of boot images for various types of computers and operating systems. The framework consists of a boot image builder application and three buckets of independently designed and implemented boot modules: a bucket of loaders, a bucket of bootstrappers, and a bucket of bootloaders.

Each bucket of boot modules relies on two interfaces: one external and one internal. Internal interfaces are standardized to allow independent development and implementation of each boot module. Each module, following the layered architecture, relies on one and implements another internal interface. A set of internal interfaces defines the framework of the Trapper OSBP.

The principal purpose of loader modules is to bring the entire boot image from boot medium into the memory. The loader should also have a processor switched from real into a protected mode. Loaders rely on the assumption that a boot image is a single continuous potentially large file both in boot medium and in memory.

Bootstrappers, in their turn, aim to switch a processor into the unified and standardized state, to collect and unify the description of a computer

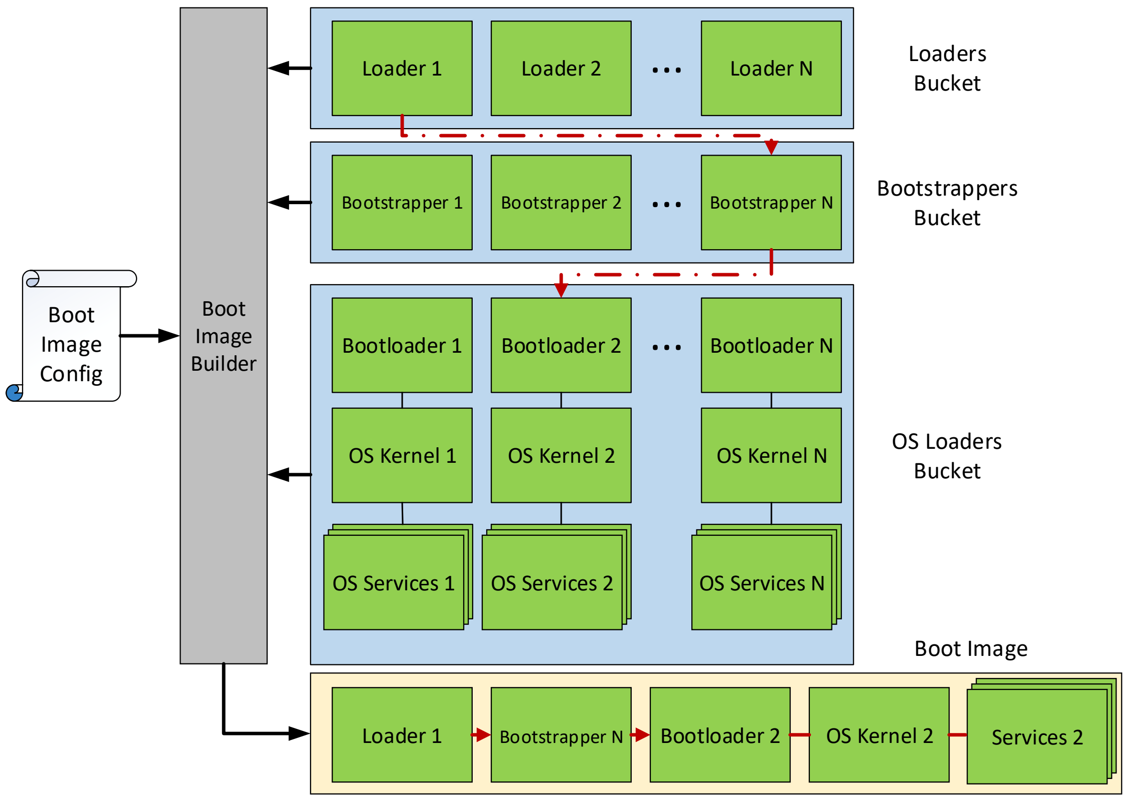

Fig. 2. Trapper OSBP architecture

system configuration. Bootloaders generally require two pieces of information from a bootstrapper. The first one is a memory map that describes the layout of the physical address space. The second one is a multiprocessor table that lists CPUs present in the system.

In contrast to the rest of the modules, Trapper expects that bootloader development is the responsibility of operating system kernel vendors. Bootloader offloads from the kernel all initialization related code. Its purposes are the initialization of the CPU, operating system kernel, and an initial set of OS services. As soon as a bootloader passes control to the first task, the entire boot process finishes, and all memory used by it becomes free.

Boot Image Builder application, as its name implies, is used by system designers to create final boot images and their installation to the specified boot devices. Boot Image Builder uses a configuration file and three buckets of boot modules, as well as a specified target as an input parameter. Guided by a configuration file, it collects required boot modules from buckets and installs them into a boot image, linking modules between each other.

Loaders

Loaders are the most numerous type of boot modules used in the OSBP. Each loader represents a particular type of boot device used in the environment of specific firmware and is used solely for bringing the entire boot image from the source storage into memory. Due to this fact, such modules are typically tiny (less than a kilobyte) programs written in assembly language.

Loaders are forced to rely on the underlying firmware interface and drivers provided by it to access the required boot medium and to capture a boot image from it. For the same reason, frequently being a part of the boot image, the loader is forced to store the size of the entire boot image, as well as offset to the bootstrapper entry point.

According to the convention declared in the loader/bootstrapper interface, when a loader passes control to the bootstrapper entry point, the boot image is already fully loaded into memory. Thus, neither bootstrapper nor bootloader does not need to access a boot device to fulfill its requirements.

Different firmware affects loader design and implementation in different ways. For example, SFI eliminates the need in the loader, because firmware itself performs all actions expected from the loader [4]. Furthermore, SFI supports only booting from system flash memory. Due to this fact, there is no variability in the types of boot devices. In its turn, UEFI assumes that the loader is a standalone UEFI application, which is capable of selecting the right boot image file on the file system and read it into memory [5].

The most complicated case is a loader for disk drives on BIOS firmware. First of all, the disk drive oriented loader consists of two separate modules: *Master Boot Record* (MBR) and *Partition Boot Record* (PBR) [6]. Both have a size of 512 bytes. The only purpose of the MBR loader is to choose boot partition and load its PBR into the memory. For that purpose, it uses a partition table that is also a part of MBR (Fig. 3). In its turn, PBR is a conventional loader that holds information about the boot image source location and size and reads it into the memory from the disk drive.

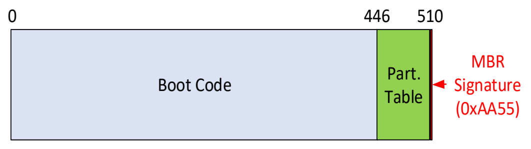

Fig. 3. Structure of the MBR

The PXE loader for BIOS-based computer systems, in turn, also implemented as a separate binary not included in the boot image. It uses PXE API [7] provided on top of BIOS API to download the boot image hosted on the remote TFTP server [8]. The boot image itself is a separate file. By design, PXE-BIOS and HDD-UEFI loaders, in contrast to the HDD-BIOS loader disk sectors-oriented design, follow the same conceptual file-oriented design but use different infrastructures.

Bootstrappers

The role of bootstrappers is to isolate operating system loaders from firmware. Therefore, all code, required to communicate with firmware, concentrates in bootstrapper modules. The bootstrappers layer relies on the interface provided by the loaders layer, and, in its turn, provides an interface for the bootloader layer. The main functions of the bootstrapper module are:

– memory map gathering,
– gathering of the list of available processors,
– disabling interrupts on the interrupt controller.

However, each implementation uses its firmware to accomplish them.

Currently, there are three well-known firmware interfaces for the IBM PC compatible computer systems: BIOS, UEFI, and SFI (Fig. 4). Therefore,

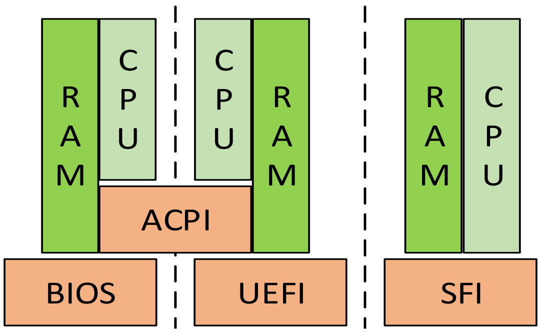

Fig. 4. Firmware and Bootstrappers

there are three bootstrapper modules available in the OSBP.

BIOS is a de facto standard for the IBM PC compatible computer systems. It operates in the 16-bit real mode of CPU and has inherited a lot of legacy features and functions. In fact, BIOS bootstrapper uses only two BIOS functions: *int 0x15-E820* [9] for querying memory map and *int 0x10-0E* for printing error messages on the screen. Bootstrapper captures a list of available processors from ACPI tables [10].

Intel has developed UEFI as a modern and standardized successor of BIOS. In contrast to BIOS, UEFI provides an advanced infrastructure for bootstrappers. It provides an interface for obtaining a memory map, but, like a BIOS, relies on the ACPI for enumeration of available processors.

While UEFI is a complete full-featured BIOS replacement for high-end computer systems (specification consists of 2899 pages) [5], SFI is a simplified BIOS successor and is extremely concise (specification contains ten pages) [4] and convenient for use in embedded systems. It implements an interface in terms of tables for both obtaining a memory map and a list of available processors.

Bootloader

Nowadays, this layer is filled only by one module for our research multikernel operating system. According to our approach, microkernel and loader are designed and developed in pairs, where bootloader offloads initialization from the kernel. As a result, microkernel does not contain pieces of "dead code" that runs only once during system startup. On the other hand, the bootloader is forced to have detailed knowledge about memory layout used by the kernel, as well as about all its internal data structures.

The bootloader of our multikernel OS (Fig. 5) divides the physical memory of a computer into zones and assigns them to processors. By doing this, it creates logical domains, each of which will be used by a separate OS node and managed by its kernel. Bootloader installs appropriate components of the boot image into the created domains to complete the OS setup. Finally, the bootloader performs startup application processors and initializes kernel data structures in parallel on all OS nodes.

The bootloader ensures that each processor is in the state expected by the microkernel. As a part of this process, it initializes page tables and enables virtual memory.

As a part of the initialization process, the bootloader setups not only the kernel but also at least one user-mode application. This application or a set of applications initialized by bootloader plays the role of the *Init* process and continues self-deployment of the operating system. Once the bootloader has its work finished, it passes control directly to the Init application, bypassing the kernel, but that Init application executes already under OS kernel control.

Boot Image Builder

The Boot Image Builder is a glue that unites all other parts of the framework. Implemented as a command-line tool to facilitate automation, it assembles the boot image from the boot modules captured from the buckets under the guidance of the XML-based configuration file passed to its input.

As mentioned earlier, the main task of the Boot Image Builder is to chain boot modules as well as OS kernels and an initial set of applications into a single image file. Guided by metadata that came with loaders, the Boot Image Builder can separate

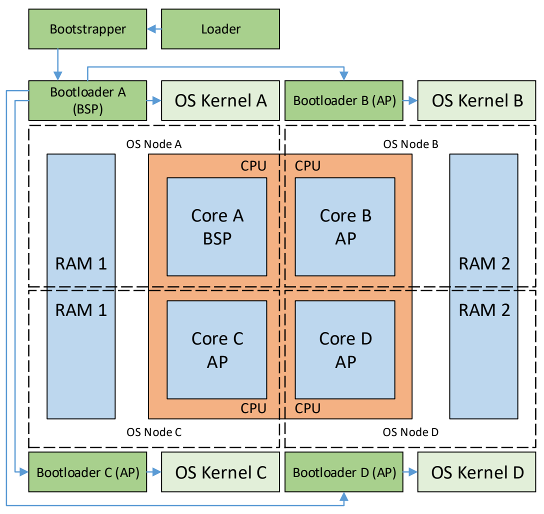

Fig. 5. Boot loading of multikernel OS

loaders from the boot image, which is the case of PXE-BIOS and GPT-HDD loaders. Additionally, it can install the final boot image into the specified disk partition. Finally, the Boot Image Builder has facilities for disk partitioning and supports both partitioning types: MBR and GPT. The MBR loader module is an exceptional component of OSBP.

At the same time, Boot Image Builder can embed a dedicated configuration portion into a boot image for later use by the bootloader. During embedding, it converts this portion of the configuration from XML into a specific binary form.

In our case, we have symmetric multikernel OS for the SMP computer system. Due to this fact, we have one type of kernel and respective bootloader, but that loader is capable of working in two modes: BSP and AP. BSP processor enables the bootloader BSP mode in which it performs deployment of the entire operating system over the application processors. BSP loader copies itself and an appropriate kernel to the memory zones assigned to the application processors to accomplish this job. Then, it starts up the application processors in such a way that instances of bootloader copies run in the AP mode, and thus, skip the OS deployment step. The Boot Image Builder assembles the boot image so that it contains only one instance of each boot module type including kernels and loaders.

Another feature of the Boot Image Builder is that it can parse executable files in PE format. Microkernel and its bootloader are built by the Microsoft C++ compiler as a single section PE executables. First, the Boot Image Builder extracts and puts into the boot image only the code sections of both files. Second, it performs "linkage" between a bootloader and a kernel. The kernel uses table-based dispatching in specific cases. The export table of kernel executable binary stores the addresses of the handler functions. In its turn, the bootloader imports these addresses through its import table. In the operating system environment, OS makes such "linkage" at run time. In the case of OSBP, the Boot Image Builder performs such linkage during boot image assembling by filling a bootloader import table by actual addresses of the handler-functions of the kernel.

Conclusion and Outlook

This paper presents the architecture of Trapper – the operating system bootstrapping package developed to facilitate booting of operating systems with alternative architectures. Trapper provides a flexible framework for building boot images targeting IBM PC compatible computer systems based on IA-32 processors. We have demonstrated our design based on three layers of boot modules and have pointed out the importance of interfaces between them for strict separation of dependencies from types of boot devices, firmware, and operating systems. In this context, we have presented an example of implementing OSBP and its components, including loaders, bootstrappers, bootloaders, and the Boot Image Builder application. We have also shown that our architecture provides a flexible environment that can be easily extended to support the additional types of boot devices, firmware, and other operating systems. Moreover, we have demonstrated how our OSBP works in the case of loading the multikernel operating system and how the full separation of initialization code works for our model of co-design and co-development of microkernel and its bootloader. Although our framework was designed and implemented for supporting IBM PC compatible computers based on processors with IA-32 and x86_64 instruction set architectures, we hope that it can be adopted to support processors with other instruction set architectures.

Acknowledgements. *The research and development have been supported by the Belarusian State University of Informatics and Radioelectronics. The authors would especially like to thank Peter Brancevich, Svyatoslav Kulikov, and Kirill Surkov for their support and assistance.*

References

1. Mazidi M., Mazidi J. *The 80x86 IBM PC and Compatible Computers: Assembly Language, Design, and Interfacing*. Prentice Hall Publ., 2003, 1024 p.

2. *IA-32 Intel Architecture Software Developer's Manual. Vol. 3: System Programming Guide*. 2002. Available at: https://pdos.csail.mit.edu/6.828/2003/readings/intelv3.pdf (accessed September 15, 2019).

3. Pelner J.M., Pelner J.A. *Minimal Intel Architecture Boot Loader: Bare Bones Functionality Required for Booting an Intel Architecture Platform*. 2010. Available at: https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf (accessed September 15, 2019).

4. Brown L. The Simple Firmware Interface. *Proc. 2009 Ottawa Linux Symp*., Montreal, Canada, 2009, pp. 55–60. Available at: https://www.kernel.org/doc/ols/2009/ols2009-pages-55-60.pdf (accessed September 15, 2019).

5. Unified EFI Forum, Inc. *Unified Extensible Firmware Interface Specification, Ver. 2.7*. 2017. Available at: http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf (accessed September 15, 2019).

6. Chelliah B., Vidyadharan D.S., Sulekha D., Thomas K.L. *Combating information hiding using forensic methodology*. Proc. 6th Intern. WDFIA, London, UK, 2011, pp. 69–75. Available at: https://pdfs.semanticscholar.org/4a48/520523bc2e3ef9b52163200ae1b1473955d8.pdf (accessed September 15, 2019).

7. *Preboot Execution Environment (PXE) Specification, Ver. 2.1*. 1999. Available at: http://www.pix.net/software/pxeboot/archive/pxespec.pdf (accessed September 15, 2019).

8. Sollins K. *The TFTP Protocol (Revision 2)*. 1992. Available at: https://tools.ietf.org/html/rfc1350 (accessed September 15, 2019).

9. Yao J., Zimmer V.J., Fleming M. *White Paper: A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS*. 2016. Available at: https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf (accessed September 15, 2019).

10. *Advanced Configuration and Power Interface Specification, Revision 2.0a*. 2002. Available at: https://www.intel.com/content/dam/www/public/us/en/documents/articles/acpi-config-power-interface-spec.pdf (accessed September 15, 2019).

11. *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*. 1999. Available at: https://courses.cs.washington.edu/courses/cse378/03wi/lectures/LinkerFiles/coff.pdf (accessed September 15, 2019).

УДК 004.451.8 Дата подачи статьи: 26.09.19
DOI: 10.15827/0236-235X.130.210-217 2020. Т. 33. № 2. С. 210–217

Trapper: программный пакет для создания загрузочных образов, предназначенных для использования в IBM PC-совместимых компьютерных системах

Е.И. Клименков [1], *аспирант, klimenkov@bsuir.by*

[1] *Белорусский государственный университет информатики и радиоэлектроники, г. Минск, 220013, Республика Беларусь*

В статье представлен обзор процесса начальной загрузки IBM PC-совместимых компьютерных систем и предложена архитектура программного пакета предназначенного для создания загрузочных образов, которые необходимы для подготовки и запуска операционной системы, и ориентированного на поддержку загрузки операционных систем с альтернативными архитектурами, такими как микроядро-ядерные, экзоядреные и многоядерные ОС.

Данная система представляется в виде трех отдельных наборов независимых загрузочных модулей, дополненных приложением Boot Image Builder, предназначенным для создания загрузочных образов операционной системы путем интеграции загрузочных модулей в единый бинарный образ и связывания их друг с другом для организации целостной цепочки загрузчиков, необходимой для приведения операционной системы в работоспособное состояние. Архитектура предлагаемой программной системы отражает принципиальные этапы процесса загрузки компьютерной системы. Каждый набор загрузочных модулей связан с определенным этапом загрузки и образует слой, решающий свой собственный четко определенный набор задач и опирающийся на четко определенные межслойные интерфейсы для строгой изоляции зависимостей от загрузочного устройства, нижележащего встроенного программного обеспечения и специфики загружаемой операционной системы. В статье представлена реализация предлагаемой архитектуры для генерации загрузочных образов, созданная для исследовательской многоядерной операционной системы, а также объясняется процесс загрузки последней.

Кроме того, предложена идея полного отделения кода инициализации от кода ядра операционной системы и его перемещения в независимый модуль загрузчика ОС. Следование этой идее приводит к исключению мертвогокода, связанного с инициализацией, из ядра ОС. В традиционных операционных системах такой код выполняется единожды при загрузке системы, однако, будучи частью исполняемого двоичного образа ядра, продолжает занимать память на всем протяжении работы компьютерной системы, вплоть до завершения ее работы.

Ключевые слова: *загрузочный образ, загрузчик, операционная система, IBMPC.*

Исследования и разработки, положенные в основу данной работы, были проведены при поддержке Белорусского Государственного Университета Информатики и Радиоэлектроники. Авторы выражают особую благодарность П. Бранцевичу, С. Куликову и К. Суркову за помощь и поддержку.

Литература

1. Mazidi M., Mazidi J. The 80x86 IBM PC and compatible computers: assembly language, design, and interfacing. Prentice Hall Publ., 2003, 1024 p.
2. Intel Corp. IA-32 Intel Architecture Software Developer's Manual. Vol. 3: System Programming Guide. 2002. URL: https://pdos.csail.mit.edu/6.828/2003/readings/intelv3.pdf (дата обращения: 15.09.2019).
3. Pelner J.M., Pelner J.A. Minimal Intel Architecture Boot Loader: Bare Bones Functionality Required for Booting an Intel Architecture Platform. 2010. URL: https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf (дата обращения: 15.09.2019).
4. Brown L. The Simple Firmware Interface. In Proc. of 2009 Ottawa Linux Symposium, Montreal, Canada, 2009, pp. 55-60. URL: https://www.kernel.org/doc/ols/2009/ols2009-pages-55-60.pdf (дата обращения: 15.09.2019).
5. Unified EFI Forum, Inc. Unified Extensible Firmware Interface Specification, Ver. 2.7. 2017. URL: http://www.uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf (дата обращения: 15.09.2019).
6. Chelliah B., Vidyadharan D.S., Sulekha D., Thomas K.L. Combating information hiding using forensic methodology. Proc. 6th Intern. WDFIA, London, UK, 2011, pp. 69–75. URL: https://pdfs.semanticscholar.org/4a48/520523bc2e3ef9b52163200ae1b1473955d8.pdf (дата обращения: 15.09.2019).
7. Intel Corp. Preboot Execution Environment (PXE) Specification, Ver. 2.1. 1999. URL: http://www.pix.net/software/pxeboot/archive/pxespec.pdf (дата обращения: 15.09.2019).
8. Sollins K. The TFTP protocol, revision 2. 1992. URL: https://tools.ietf.org/html/rfc1350 (дата обращения: 15.09.2019).
9. Yao J., Zimmer V.J., Fleming M. White Paper: A Tour Beyond BIOS Memory Map and Practices in UEFI BIOS. 2016. URL: https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Memory_Map_And_Practices_in_UEFI_BIOS_V2.pdf (дата обращения: 15.09.2019).
10. Compaq Computer Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., Toshiba Corp. Advanced Configuration and Power Interface specification, revision 2.0a. 2002. URL: https://www.intel.com/content/dam/www/public/us/en/documents/articles/acpi-config-power-interface-spec.pdf (дата обращения: 15.09.2019).
11. Microsoft Corp. Microsoft Portable Executable and Common Object File Format Specification, revision 6.0. 1999. URL: https://courses.cs.washington.edu/courses/cse378/03wi/lectures/LinkerFiles/coff.pdf (дата обращения: 15.09.2019).

Для цитирования

Клименков Е.И. Trapper: программный пакет для создания загрузочных образов, предназначенных для использования в IBM PC-совместимых компьютерных системах // Программные продукты и системы. 2020. Т. 33. № 2. С. 210–217. DOI: 10.15827/0236-235X.130.210-217.

For citation

Klimiankou Y.I. Trapper: an operating system bootstrapping package for IBM PC compatible computer systems. *Software & Systems*. 2020, vol. 33, no. 2, pp. 210–217 (in Russ.). DOI: 10.15827/0236-235X.130.210-217.