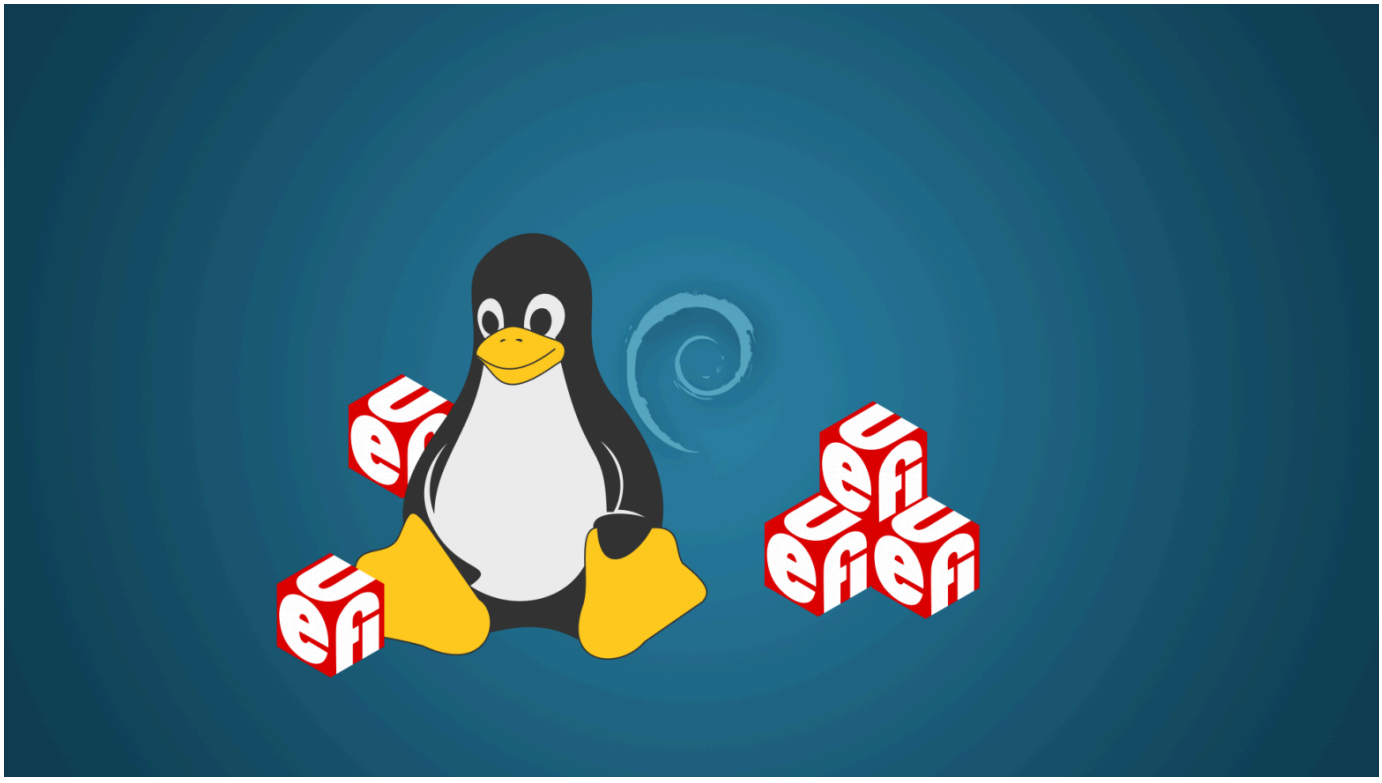
artyomsoft
6 мар в 12:01

Пишем свой загрузчик операционной системы Linux

Средний 23 мин 24K

Блог компании RUVDS.com, UEFI*, Программирование*, Разработка под Linux*, Системное программирование

Туториал



Меня давно интересовал вопрос, насколько сложно написать собственный загрузчик операционной системы. Я не говорю о простой программе, выводящей «Hello, World!», а о полноценном загрузчике, который передаёт управление от встроенного программного обеспечения компьютера ядру операционной системы. Современные загрузчики представляют собой сложные программы, способные загружать множество операционных систем различными способами, учитывая массу нюансов, связанных с программным и

аппаратным обеспечением. Читая их исходный код, легко утонуть в деталях и потерять понимание сути и реализации.

Я решил начать изучение с максимально простого подхода, постепенно усложняя задачи, экспериментируя и получая новые знания. Если мне удалось вас заинтересовать, добро пожаловать под кат.

Введение

Рассказать о том, как написать загрузчик, — задача не из тривиальных. Это связано с тем, что затрагивается множество связанных и несвязанных тем и требуется хотя бы базовое понимание следующего:

- архитектуры компьютера;
- режимов работы процессора;
- синтаксиса языков C и ассемблера;
- указателей в C (без них здесь не обойтись);
- основ разработки UEFI-приложений;
- организации работы с оперативной памятью;
- соглашений о вызовах функций;
- работы компоновщика;
- форматов файлов ядра операционной системы (bzImage, PE32+/COFF).

Естественно, что полноценно охватить все эти темы в статье невозможно, но я старался изложить материал так, чтобы читатели с минимальными знаниями в этих областях могли понять основы написания загрузчика и попробовать реализовать его самостоятельно. Часть информации можно извлечь из загрузчика для учебных целей asbootsap, написанного мной. Я уверен, что можно написать более безопасный и надёжный код. Но моя цель заключалась в написании загрузчика, способного загружать Linux и содержащего минимум программного кода. Поэтому я установил для себя следующие ограничения для загрузчика:

- он должен быть UEFI-приложением для архитектуры x86_64,
- он должен уметь загружать современные ядра Linux для этой архитектуры в форматах PE32+ и bzImage,
- он должен загружать ядра и initramfs, находящиеся на том же ESP-разделе, что и сам загрузчик,

- он не должен поддерживать Secure Boot.

Вас не должны пугать эти ограничения, так с этим загрузчиком можно загрузить большинство известных дистрибутивов Linux. Если вас испугали термины в начале, но есть желание разобраться, не бойтесь, по прочтению статьи, многие перестанут быть набором непонятных символов.

Надеюсь, вы получите удовольствие от собственноручно переписанного с моего примера загрузчика, а после получения новых знаний работа загрузчика не будет казаться магией.

Если вы хотите проверить всё на практике, вам необходимо будет установить какой-нибудь дистрибутив Linux.

Как происходит дальнейшая загрузка ядра Linux, я не рассматриваю, одна из целей моей статьи — изложить, как происходит передача управления от UEFI ядру операционной системы. А начну я с того, как можно загрузить Linux без использования привычного многим загрузчика, такого как, например, GRUB.

Основы UEFI или минимальный набор знаний для написания загрузчика

UEFI

Unified Extensible Firmware Interface (UEFI) — это спецификация, разработанная Unified EFI Forum, которая описывает интерфейс между операционной системой и прошивкой (firmware) компьютера. Спецификация накладывает определённые ограничения на архитектуру разрабатываемых приложений. Хотя UEFI не зависит от языка программирования, описания структур и функций в спецификации приведены на языке C.

Как правило, для спецификации существует *reference implementation*, UEFI не является исключением и имеет reference implementation TianoCore EDK II. Она представляет собой среду разработки, позволяющую создавать драйвера и приложения для UEFI-систем. Разрабатывать и отлаживать UEFI-приложения удобнее в виртуальных машинах, поэтому существует специальная реализация UEFI-Firmware для виртуальных машин Qemu и KVM — OVMF.

TianoCore EDK II может быть сложной для понимания и освоения. Для разработки более простых UEFI-приложений можно использовать легковесную среду gnu-efi, которую я и буду использовать в статье.

Спецификация UEFI достаточно объёмный документ. Но я изложу суть, которая позволит вам написать свой загрузчик.

Можно сказать, что UEFI имеет объектно-ориентированную архитектуру. Основу составляют такие понятия, как *Protocol*, *Protocol Interface*, *Handle*, *System Table*, *Boot Services*, _Runtime Services. Для меня сложно было с первого раза понять суть Protocol, Protocol Interface, Services в UEFI, так как термины употребляются в немного другом контексте, отличном от привычного.

О режимах работы процессора архитектуры x86-64

Исторически так сложилось, что при включении питания процессор находится в так называемом Real Mode. В этом режиме используется 16-битная сегментная адресация, и доступно только 1 Миб памяти, нет защиты памяти, нет поддержки многозадачности, все адреса являются физическими. Для работы операционной системы с 32-битным ядром необходимо переключиться в так называемый *Protected Mode*, а с 64-битным в *Long Mode*, которые лишены этих недостатков. Информация о том, как переключаться в эти режимы нам не понадобится, так как в момент передачи управления загрузчику UEFI сделает это за нас. Эта информация была необходима, когда использовалась передача управления загрузчику из более старого 16-битного BIOS. При 64-битном UEFI процессор будет находиться в Long Mode.

UEFI Images

Исполняемый код в UEFI хранится в файлах формата PE32+/COFF. Существует 3 вида UEFI Images:

- UEFI Application,
- UEFI OS Loader,
- UEFI Driver.

По сути UEFI OS Loaders — это подвид UEFI Application, задача которого передать управление операционной системе, предварительно вызвав ExitBootServices(). UEFI Driver мы рассматривать не будем, главное его отличие от UEFI Application, что он остается резидентным в памяти после возврата из точки входа.

Загрузка UEFI Image и загрузка файла

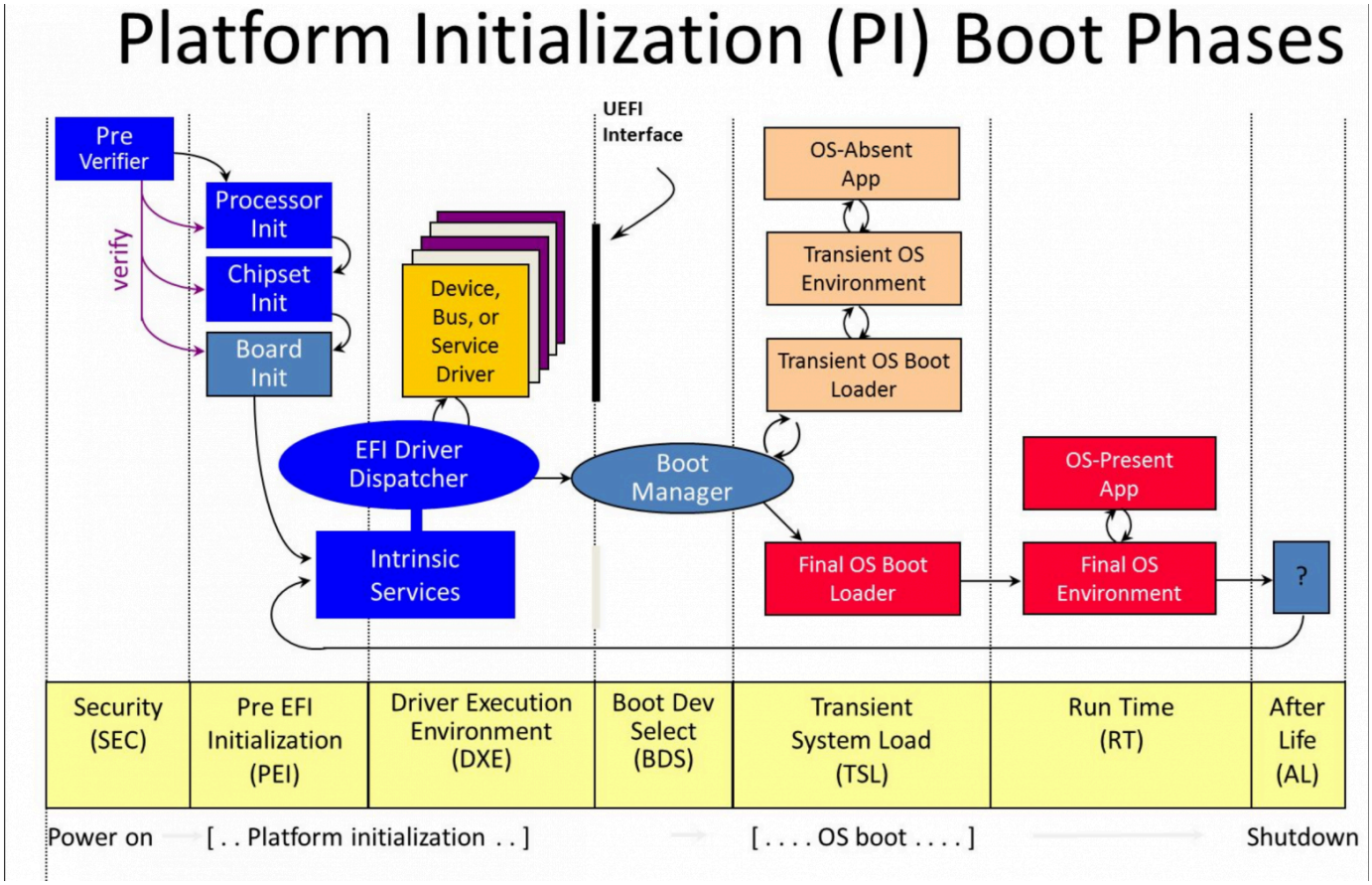
В UEFI можно загрузить как UEFI Image, так и файл, и это важно различать. Когда вы загружаете UEFI Image с помощью функции *LoadImage()*, система автоматически выделяет

оперативную память, анализирует PE-заголовок и размещает содержимое образа в соответствующих областях памяти.

Загрузка файла представляет собой более простую операцию, но от разработчика требуется больше усилий. В этом случае необходимо самостоятельно выделить память для содержимого файла, а затем использовать функции для работы с файлами, чтобы считать данные с устройства и разместить их по адресу выделенной памяти. В нашем случае устройством будет файловая система на разделе ESP.

Фазы загрузки UEFI

Ранее я объяснил простыми словами, как осуществляется загрузка. Ниже приведу классический рисунок, иллюстрирующий фазы загрузки UEFI-окружения.



Фазы загрузки UEFI

Для нас представляет интерес только фаза *Transient System Load* (TSL). В этой фазе осуществляется передача управления от встроенного программного обеспечения операционной системе.

OS Loader Image

OS Loader Image — подвид приложений UEFI, задача которого — передать управление от UEFI ядру операционной системы. Как правило, должен находиться на ESP-разделе диска,

но может располагаться и в компьютерной сети или, более экзотический вариант, прошит в микросхеме материнской платы.

Как запустить OS Loader Image? Самый простой способ — назвать его BOOTX64.EFI и поместить в директорию на разделе ESP. Когда вы выберете в настройках UEFI загрузку с диска, на котором расположен этот раздел, UEFI запустит OS Loader Image. Второй способ предполагает использование EFIShell — UEFI-приложения, предоставляющего интерфейс командной строки к UEFI, напоминающий bash или другую подобную оболочку. Также можно прописать информацию о загрузчике в UEFI Boot Manager.

System Table

UEFI Image имеет точку входа. Точка входа — это функция с двумя параметрам: Handle самого UEFI-приложения и указатель на System Table. System Table является важной структурой, с помощью которой UEFI-приложение может взаимодействовать с UEFI-окружением, и содержит указатели на интерфейсы ввода/вывода, Boot Services и Runtime Services. Можно сказать, что System Table является корнем иерархии UEFI. Чтобы вам было понятнее, приведу исходный код её определения из файла efiapi.h.

```
typedef struct _EFI_SYSTEM_TABLE {
EFI_TABLE_HEADER Hdr;

CHAR16 *FirmwareVendor;
UINT32 FirmwareRevision;

EFI_HANDLE ConsoleInHandle;
SIMPLE_INPUT_INTERFACE *ConIn;

EFI_HANDLE ConsoleOutHandle;
SIMPLE_TEXT_OUTPUT_INTERFACE *ConOut;

EFI_HANDLE StandardErrorHandle;
SIMPLE_TEXT_OUTPUT_INTERFACE *StdErr;

EFI_RUNTIME_SERVICES *RuntimeServices;
EFI_BOOT_SERVICES *BootServices;

UINTN NumberOfTableEntries;
EFI_CONFIGURATION_TABLE *ConfigurationTable;

} EFI_SYSTEM_TABLE;
```

Services

Services в UEFI немного отличаются от сервисов, которым мы привыкли в программировании веб-приложений или системном программировании под Windows. Сервисы в UEFI — это функции, выполняющие системные задачи. Например, найти Handle объекта по идентификатору поддерживаемого протокола или выделить/освободить память.

Существуют Boot Services (доступные в фазе TSL) и Runtime Services (доступные как в фазе TSL, так и после её завершения). Boot Services и Runtime Services — это ключевые компоненты инфраструктуры UEFI, предоставляющие доступ к функциональности прошивки и обеспечивающие работу драйверов и приложений.

Я не использовал в своём загрузчике Runtime Services, но при желании читатель может их также попробовать их использовать, например, добавить отображение текущего времени в загрузчике.

Boot Services и Runtime Services содержат только базовые функции UEFI, остальные, располагающиеся в приложениях и драйверах, доступны через механизм Handle/Protocol/Protocol Interface. UEFI является *extensible* именно благодаря ему.

Handle, Protocol и Protocol Interface

Сущности, с которым мы можем взаимодействовать в UEFI, имеют уникальный идентификатор, называемый *Handle*. Handle находят в UEFI-окружении по идентификатору протокола при помощи функций Boot Services *LocateHandle* или *LocateHandleBuffer*. Для изменения состояния сущности следует запросить её Interface, используя функции *HandleProtocol* или *OpenProtocol*. Каждый UEFI Protocol имеет уникальный идентификатор (GUID) и определение Protocol Interface Structure. Когда вы запрашиваете Interface, вы получаете экземпляр Protocol Interface Structure, содержащей данные и указатели на функции. Работать с состоянием объекта UEFI можно, изменяя эти данные и вызывая функции.

Memory Map

Загрузчик или ядро операционной системы должны знать о том, как распределены адреса оперативной памяти, какую память можно использовать, а какую трогать нельзя. Эта информация хранится в структуре, называемой *Memory Map*. Для получения Memory Map необходимо вызвать системную функцию GetMemoryMap в случае UEFI, или инициировать программное прерывание INT 0x15 (System BIOS Services) с AX=0xE820 (Query System Address Map) в случае BIOS. Memory Map, полученная с использованием прерывания BIOS, называется *E820 Memory Map*. E820 Memory Map будет необходима нам для

загрузки Linux, но получать мы её будем путём преобразования из EFI Memomory Map и прерывание BIOS вызывать не будем.

Загрузка операционной системы

О процессе загрузки операционной системы много написано в книгах, статьях и интернете, снято множество видеороликов, но я хочу выделить самое важное, нужное для создания нашего загрузчика. Загрузка может немного различаться в зависимости от архитектуры компьютера, встроенного программного обеспечения материнской платы, операционной системы. Поэтому далее в статье по умолчанию будет подразумеваться архитектура x86-64, ПО материнки UEFI и операционная система Linux.

1. UEFI инициализируется и проверяет оборудование (процессор, оперативную память и т.д.).
2. UEFI ищет загрузчик операционной системы на подключённых устройствах (например, SSD или HDD). Обычно это файл в разделе ESP. По умолчанию для 64-битных систем путь к загрузчику стандартизирован: .EFI для 64-битных систем. Можно разместить его и по другому пути, но для этого необходимо прописать *загрузочную запись* в UEFI.
3. После нахождения загрузчика UEFI передаёт управление ему. Загрузчик подготавливает систему к загрузке ядра операционной системы.
4. Далее управление от загрузчика передается ядру операционной системы, которая продолжает дальнейшую загрузку.

Загрузчик является промежуточным звеном между UEFI и операционной системой, так как его код выполняется до загрузки последней, он не может использовать привычные функции из операционной системы, сборка его исходного кода, а также отладка отличаются.

Можно загрузить операционную систему без него, но это менее гибкий вариант и существует ряд особенностей.

Загрузка Linux без использования загрузчика

Для вас может стать открытием, что можно загрузить Linux без использования загрузчика на системах UEFI. Да, это действительно возможно, если ядро Linux скомпилировано с установленным параметром CONFIG_EFI_STUB. Хочу вас обрадовать, в современных ядрах этот параметр, как правило, включён.

Для текущего ядра Linux узнать включён этот параметр или нет, можно командой:

```
cat /boot/config-$(uname -r) | grep CONFIG_EFI_STUB
```

Для загрузки Linux необходимо следующее:

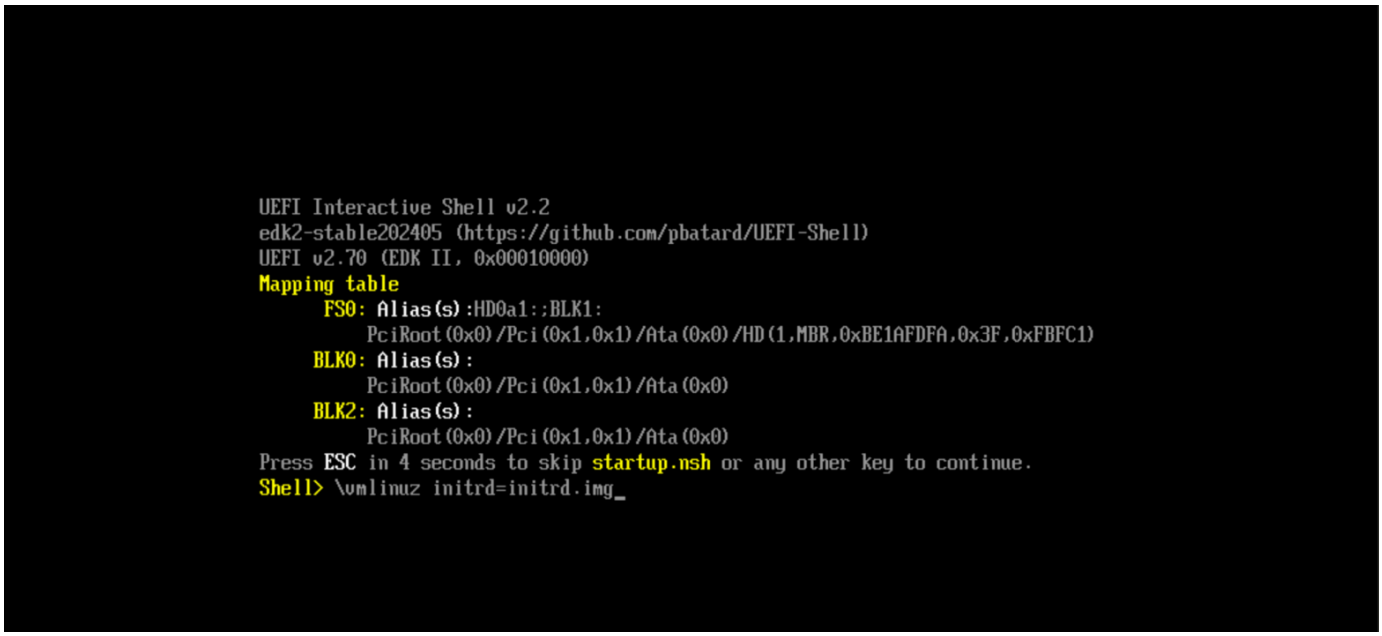
1. Файл ядра Linux.
2. Файл с образом начальной корневой файловой системы.
3. Корневая файловая система.
4. Параметры командной строки ядра (kernel command-line parameters).

Мне известно о четырёх способах загрузки Linux без загрузчика:

1. Использование командной строки UEFI Interactive Shell.
2. Использование файла startup.nsh.
3. Добавление загрузочной записи в UEFI NVRAM с помощью команды bcfg в UEFI Interactive Shell.
4. Добавление загрузочной записи в UEFI NVRAM с помощью команды efibootmgr в Linux.

Использование командной строки UEFI Interactive Shell

```
\vmlinuz initrd=/initrd.img root=UUID=7023590e-426d-4087-bb7d-4bc6fd1bbc7a quiet splash
```



Использование файла startup.nsh

В этом файле можно прописать команду по загрузке Linux, которая используется в первом способе. Её не нужно вводить постоянно, она будет выполняться каждый раз при включении компьютера. Кодировка файла должна быть ASCII.

Добавление загрузочной записи в UEFI NVRAM при помощи команды UEFI Interactive Shell bcfg

1. Переходим в корень ESP-раздела:

```
fs0:\
```

2. Выводим загрузочные записи:

```
bcfg boot dump
```

3. Добавляем загрузочную запись:

```
bcfg boot add 2 fs0:\vmlinuz "My Linux"
```

4. Параметры командной строки для передачи bcfg должны располагаться в файле, поэтому создаем файл options.txt:

```
edit options.txt
```

5. Вводим содержимое файла и сохраняем, нажав клавишу F2, а потом F3:

```
initrd=/initrd.img root=UUID=7023590e-426d-4087-bb7d-4bc6fd1bbc7a quiet splash
```

6. Добавляем содержимое командной строки в загрузочную запись:

```
bcfg boot -opt 2 fs0:\options.txt
```

7. Если хотим удалить загрузочную запись:

```
bcfg boot rm 2
```

Добавление загрузочной записи в UEFI NVRAM при помощи команды Linux efibootmgr

efibootmgr позволяет добавлять и управлять загрузочными записями в UEFI NVRAM, обеспечивая возможность загрузки Linux без загрузчика.

```
sudo efibootmgr -c -d /dev/sda -p 1 -L "My Linux" -l '\vmlinuz' -u 'root=UUID=7023590e-
```

efibootmgr создаёт загрузочную запись с названием My Linux на первом разделе диска /dev/sda. В приведенном выше примере UEFI загружает файл vmlinuz, находящийся в корне файловой системе раздела. В качестве параметров ядру передадутся: initrd, root, quiet splash Параметр intird указывает расположение файла начальной корневой файловой системы. Параметр root указывает на расположение корневой файловой системы. Расположение корневой файловой системы можно задать различными способами: например, указав имя блочного устройства раздела, где она располагается (/dev/sda1), а можно, указав идентификатор раздела или файловой системы (это более предпочтительный способ). Тут используется идентификатор файловой системы. Идентификатор файловой системы, расположенной на разделе, можно узнать при помощи команды:

```
blkid -s UUID -o value /dev/nvme0n1p1
```

А имена присутствующих разделов можно узнать при помощи команды:

```
lsblk -o name -lpn
```

Параметры quiet и splash можно не использовать, они нужны, если вы хотите минимизировать вывод сообщений при загрузке ядра.

Особенности и ограничения загрузки Linux без использования загрузчика

Обратите внимание, что:

1. UEFI Interactive Shell может отсутствовать во многих прошивках UEFI. В таком случае вам нужно скачать UEFI Shell с сайта, переименовать его в BOOTX64.EFI и поместить в директорию на ESP-разделе.
2. Запись в UEFI NVRAM является потенциально рискованной операцией. Вы должны чётко понимать, что вы делаете и зачем. Также неверно реализованный драйвер или команда могут привести к повреждению содержимого SPI-чипа на материнской плате. Кроме того, частые записи на чип ускоряют его износ.
3. Загрузить без загрузчика можно только, если ядро Linux скомпилировано с определёнными опциями. Большинство современных ядер компилируется с ними, так что, скорее всего, у вас проблем не возникнет.

Я советую экспериментировать с загрузкой, используя виртуальную машину qemu. В этом вам может помочь мой небольшой проект для создания образа диска с ESP-разделом и небольшим дистрибутивом на основе Linux. Более подробно, как создавать дистрибутивы Linux я рассматривал в своей статье

Поэкспериментировав с загрузкой Linux без загрузчика, перейдём к написанию собственного.

Загрузчик операционной системы Linux

Что делает загрузчик операционной системы Linux? Он, следуя заданным настройкам, находит доступное для загрузки ядро и соответствующий файл образа начальной файловой системы (initramfs/initrd), загружает их в оперативную память и запускает ядро, передавая ему параметры командной строки, включающие информацию о корневой файловой системе и другие настройки.

Как запустить загрузчик? В случае UEFI — загрузчик является UEFI-приложением. Запустить загрузчик вы можете:

- запустив из UEFI Intercative Shell,
- прописав загрузчик в загрузочную запись для него в NVRAM
- переименовать его в BOOTX64 и поместить в на ESP-разделе.

Правила передачи управления от загрузчика к ядру операционной системы называются *протоколом загрузки*.

Формат файла ядра Linux для архитектуры x86-64

Ядро Linux представляет собой ELF-файл, однако загрузчики на архитектурах x86-64 не могут работать с этим форматом (по крайней мере, я не встречал). Ядро на архитектуре x86-64 упаковывается в файл формата bzImage. Что будет содержать файл bzImage, задается на этапе компиляции ядра. Как правило, это:

- заголовок ядра,
- код начального запуска ядра из реального режима,
- код для распаковки ядра
- сжатое ядро.

Большинство современных ядер имеют EFIStub — специальный код, который позволяет UEFI рассматривать ядро Linux как UEFI-приложение.

Для нас важно следующее:

- где находится описание заголовка ядра,
- где находятся точки входа для запуска ядра.

Заголовок ядра занимает не более двух секторов (1024 байтов) и в зависимости от конфигурации ядра, заданного при его компиляции, может различаться. Например, если ядро скомпилировано с параметром CONFIG_EFI_STUB, в заголовке будут присутствовать PE и COFF заголовки.

Ядро Linux в формате bzImage имеет несколько точек входа:

- для реального режима,
- для защищённого режима,
- для long режима,
- для EFI Handover,
- для UEFI Firmware efi_pe_entry.

Нас интересуют только последние три.

Протоколы загрузки операционной системы

Я долгое время считал, что для загрузки Linux используется Multiboot Protocol, но это не так. На архитектуре x86-64, как правило, применяется Linux Boot Protocol. Статья с описанием Linux Boot Protocol является отправной точкой для погружения в подробности магии загрузки. Некоторые вещи, изложенные в статье по Linux Boot Protocol, для меня до сих пор остаются загадкой, а написать загрузчик, используя только информацию, изложенную там, вряд ли получится. Несмотря на это, в той статье много информации касается загрузки на устаревших системах с Legacy BIOS, она значительно помогла разобраться в теме. Я рассмотрю *64-bit Linux Boot Protocol*, *EFI Handover* и, хотя он там не описан, самый простой в программировании протокол *Chainload*.

Chainload

Суть протокола заключается в том, что современные ядра Linux являются валидными UEFI-приложениями. Это позволяет загрузчику загружать и выполнять UEFI-образ приложения, используя API, предоставляемые UEFI. Initramfs загружается в оперативную память не загрузчиком, а самим ядром Linux, при этом указание, какой initramfs использовать, передаётся одним из параметров командной строки Linux. В рамках данного протокола можно загружать только те ядра, которые содержат поддержку EFIStub.

EFI Handover

Протокол EFI Handover подразумевает, что загрузчик помещает содержимое ядра и initramfs в оперативную память, предварительно разобрав заголовок файла ядра Linux в формате *bzImage*. Для передачи управления Linux загрузчик должен вызвать функцию, адрес которой записан в поле *handover_offset* заголовка. Для архитектуры x86-64 абсолютный адрес функции вычисляется следующим образом:

```
handover_function_address = kernel_loading_address + handover_offset + 512
```

Где:

- kernel_loading_address — адрес, по которому загрузчик загрузил ядро Linux,
- hadover_offset — смещение внутри этого файла, где располагается точка входа для загрузки по протоколу EFI Handover,
- 512 — дополнительное смещение в 512 байт, если используется архитектура x86-64.

Функция требует три параметра:

- дескриптор EFI-приложения,
- указатель на EFI System Table
- указатель на заполненную структуру *boot_params*.

Протокол EFI Handover считается устаревшим, но я его привёл, так как считаю его заслуживающим рассмотрения.

64-bit Linux Boot Protocol

64-bit Linux Boot Protocol самый сложный из трёх, из рассматриваемых. Загрузчик не только помещает ядро и initramfs в оперативную память и заполняет структуру *boot_params*, но и должен выполнить дополнительные действия:

- Сконфигурировать *framebuffer*.
- Подготовить *E820 Memory Map* из *UEFI Memory Map*.
- Подготовить информацию о UEFI-окружении.
- Осуществить выход из UEFI Preboot Environment, вызвав *ExitBootServices()*.

- Вызвать функцию, адрес которой для архитектуры x86-64 вычисляется как:

```
boot64_function_address = kernel_loading_address + 512
```

Написание загрузчика Linux

Написание загрузчика подразумевает разработку программы, работающую в UEFI-окружении, использующую сервисы и протоколы UEFI и реализующую протокол загрузки ядра, требуемый операционной системой (не путать протокол загрузки ОС с протоколом UEFI).

Разрабатывать загрузчик с нуля только по спецификациям, наверное, можно, но проще подсмотреть часть кода в существующих загрузчиках с открытым исходным кодом. Программный код всё-таки лучше объясняет то, что написано в спецификации, так как исключает неоднозначности. Поэтому я поверхностно изучил программный код известных мне загрузчиков.

- GRUB 2,
- rEFInd,
- systemd-boot,
- SimpleBoot,
- EfiLinux,
- Limine,
- ELILO.

За основу мной был взят код EfiLinux (давно написанного примера загрузчика). Он поддерживает загрузку по протоколу EFI-Handover и Linux Boot Protocol (для старых ядер Linux). Моя модификация его кода для загрузки по протоколу Linux Boot Protocol для новых ядер не помогла, что меня зацепило, и мной был написан свой загрузчик, который был лишён этого недостатка. Очень помогли исходные коды Limine (он дал уверенность в том, что UEFI-загрузчик с использованием Linux Boot Protocol возможен) и rEFInd (я разобрался, как передавать параметры ядра при загрузке по протоколу Chainload).

Я разрабатывал загрузчик итеративно, но в статье привожу только полученный результат.

Первая версия моего простейшего загрузчика без поддержки конфигурационного файла, который как по волшебству загрузил Debian, установленный на моём ноутбуке, отсутствует,

так как это менее презентабельно и удобно с точки зрения пользователя. Это немного усложнило код, но загрузчик теперь можно использовать не только в учебных целях, а и для загрузки реальных дистрибутивов Linux, хотя и с ограничениями.

К параметрам, поддерживаемым моим загрузчиком относятся:

- выбранный протокол загрузки Linux,
- местоположение в файловой системе ESP-раздела файла ядра Linux,
- местоположение в файловой системе ESP-раздела файла initramfs,
- параметры командной строки ядра Linux.

Алгоритм работы моего загрузчика Linux

Основной алгоритм работы загрузчика следующий:

1. После передачи управлению загрузчику, он находит и разбирает конфигурационный файл с параметрами загрузки.
2. Если протокол загрузки Linux не задан, пользователю отображается меню, где он может выбрать протокол загрузки.
3. В зависимости от выбранного протокола загрузки выполняется один из следующих алгоритмов.

Алгоритмы работы загрузчика для различных протоколов загрузки

EFI chainload

1. Определяется путь к устройству, с которого был загружен образ загрузчика (в нашем случае этим устройством будет файловая система на ESP-разделе).
2. Загружается в память ядро Linux содержащее EFIStub, как образ UEFI-приложения.
3. Параметры командной строки Linux преобразуются в LoadOptions. (Путь к initramfs задаётся параметром командой строки initrd).
4. Выполняется запуск ядра Linux как образа UEFI приложения. (Параметры командной строки ядра Linux хранятся в LoadOptions).

EFI Handover

1. Определяется путь к устройству, с которого был загружен образ загрузчика.
2. Читается заголовок в файле ядра.
3. Выделяется память и записывается в неё ядро операционной системы (без загрузочных секторов и кода загрузки из реального режима).
4. Выделяется память и записывается в неё файл initramfs.
5. Выделяется и заполняется память для командной строки.
6. Определяется точка входа для EFI Handover.
7. Осуществляется передача управлению ядру путём обращения к точке входа.

64-bit Linux Boot Protocol

1. Определяется путь к устройству, с которого был загружен образ загрузчика.
2. Читается заголовок в файле ядра.
3. Выделяется память и записывается в неё ядро операционной системы (без загрузочных секторов и кода загрузки из реального режима).
4. Выделяется и заполняется память для командной строки.
5. Формируется информация об используемом фрейм буфере.
6. Из UEFI memory map получается memory map в формате E820.
7. Формируется информация об UEFI окружении.
8. Вычисляется точка входа для 64-bit Linux Boot Protocol.
9. Настраивается GDT и сегментные регистры (загрузка выполнилась и без этого шага, но почему-то большинство существующих загрузчиков его выполняют, поэтому я выполнил этот шаг. Буду признателен, если кто-то мне объяснит, зачем она настраивается).
10. Осуществляется передача управлению ядру путём обращения к точке входа.

Замечание по работе загрузчика

Выше был приведен укрупнённый алгоритм работы загрузчика, без учёта граничных условий и обработки ошибок. Часть ошибок и граничных условий я учёл в своём программном коде, часть не учитывал с целью упрощения чтения кода (оставил пытливому читателю :)). Несмотря на простоту алгоритма, каждый из шагов требует понимания основ UEFI, программирования на языке C или ассемблера. Если у вас есть базовое понимание, как C работает с памятью, и арифметики указателей, разобраться с кодом будет проще.

Создание EFI-приложений с использованием gnu-efi

Создание EFI-приложений с помощью gnu-efi несложный процесс, однако нужно понимание некоторых вещей.

Соглашение о вызовах

Мы уже рассматривали с вами, как осуществляется взаимодействие UEFI-приложения и UEFI-прошивки. Но не учитывали так называемые соглашения о вызовах функций.

Соглашение о вызове определяет, как передаются аргументы в функцию, как возвращаются результаты, кто отвечает за очистку стека и как используются регистры процессора. Эти правила обеспечивают совместимость между вызывающей стороной (caller) и вызываемой функцией (callee), особенно в случае, если они написаны на разных языках программирования или компилируются разными компиляторами. Наверное, лучше всего понять соглашение о вызове функции, если вы попытаетесь написать функцию на языке ассемблера и вызвать её из кода, написанного на си, но это не тема моей статьи.

Так исторически сложилось, что в разных операционных системах на разных архитектурах используются разные соглашения о вызовах функций. Соглашения о вызове функций являются частью Application Binary Interface (ABI). UEFI, хотя и не является операционной системой, также предъявляет требования к соглашению о вызовах функций. Для разных архитектур (Instruction Set Architecture) в UEFI используются разные соглашения о вызовах, для x86-64 — это Microsoft’s 64-bit calling convention.

Соглашения нужны для того, чтобы компилятор корректно сгенерировал объектный код для вызова функции. Интересно, но функции, внутренние для UEFI-приложения могут иметь любое соглашение о вызове. Соблюдение соглашения о вызове важно только для функций UEFI-приложения, которые вызывает UEFI-прошивка, и для функций UEFI-прошивки, которые вызывает UEFI-приложение.

Нам важно знать о соглашениях о вызовах функций, так как мы используем компилятор gcc для архитектуры x86-64, который по умолчанию использует соглашение отличное от Microsoft’s 64-bit calling convention.

Стадии построения приложения

1. Препроцессинг (обработка препроцессом заголовочных файлов *.h).
2. Компиляция (получение объектных файлов *.o).

3. Компоновка (получение статической/разделяемой библиотеки или исполняемого файла).

При создании EFI-приложения при помощи gnu-efi добавляется ещё одна стадия — преобразование разделяемой библиотеки в файл формата .efi. Это справедливо при использовании компилятора gcc, использование clang позволяет избежать этой стадии, но я использовал gcc в учебных целях.

Разработка EFI-приложения отличается от разработки привычных пользовательских приложений, так как оно запускается не в операционной системой, а в EFI-окружении. Например:

- Оно не использует стандартную библиотеку C (glibc), так как последняя использует системные вызовы операционной системы, недоступные нам в EFI-окружении.
- Отладка EFI-приложений требует дополнительных настроек и инструментов.
- Нам доступна вся физическая оперативная память компьютера, поэтому нужно быть более аккуратным при работе с ней.
- Приложение работает в однозадачном режиме и используется только одно ядро процессора.

Компоновка и скрипты компоновки

Компоновка подразумевает, что несколько объектных файлов объединяются в библиотечный или исполняемый файл. Обычно существуют стандартные правила, как это делать, но в случае использования gnu-efi необходим пользовательский скрипт компоновки, предоставляемый gnu-efi. Я не буду рассматривать подробности, вам только важно знать, что нужно использовать этот файл при компоновке.

Содержимое пакета gnu-efi

Пакет gnu-efi содержит в себе различные файлы, приведу те, о которых нам необходимо знать:

- заголовочные файлы (*.h для работы с UEFI API и вспомогательными функциями)
- libefi.a — реализация UEFI API,
- libgnuefi.a — реализация вспомогательных функций,
- crt0-efi-x86_64.o — стартовый код EFI-приложения,

- elf_x86_64_efi.lds — пользовательский скрипт компоновки.

Расположение библиотечных файлов, скрипта компоновки и заголовочных файлов gnu-efi

Пакет gnu-efi не содержит *.pc файла, а значит, не поддерживается утилитой pkg-config, поэтому расположение заголовочных файлов и библиотек нужно будет явно указывать компилятору и компоновщику. На моём дистрибутиве они располагались в /usr/include/efi и /usr/lib соответственно. Вы всегда можете найти расположение, используя команду:

```
find . -name <имя файла>
```

Алгоритм получения EFI-приложения с использованием gcc и gnu-efi

1. Установить необходимые пакеты для разработки, если они отсутствуют.
2. Подготовить исходный код приложения.
3. Выполнить компиляцию исходного кода в объектный код.
4. Выполнить компоновку объектных файлов, библиотек gnu-efi, стартового кода EFI-приложения в разделяемую библиотеку при помощи пользовательского скрипта компоновки.
5. Преобразовать полученную библиотеку в EFI при помощи команды objcopy.

Реализация загрузчика

Ну вот, наконец, я изложил достаточно материала, и вы можете рассмотреть исходный код моего загрузчика.

- main.c — содержит точку входа в UEFI-приложение.
- bootloader.c — передаёт управление выбранному интерактивно или прописанного в конфигурационном файле протоколу загрузки.
- chainload.c — содержит функции, специфичные для загрузки при помощи протокола chainload.
- efihandover.c — содержит функции, специфичные для загрузки при помощи протокола EFI Handover.

- linuxboot64.c — содержит функции, специфичные для загрузки при помощи 64-bit Linux Boot Protocol.
- common.c — содержит функции, общие для EFI Handover Protocol и 64-bit Linux Boot Protocol.
- gdtutils.asm — содержит код для настройки GDT.
- memory.c — содержит функции, упрощающие работу с памятью.
- configparser.c — содержит функции для разбора конфигурационного файла.
- filesystems.c — содержит функции, упрощающие работу с файлами.
- debugutils.c — содержит функции, упрощающие отладку.

Подробно рассматривать файлы с исходным кодом, не вижу смысла, так как вы можете посмотреть и изучить их сами.

Сборка моего загрузчика

1. Выбрать компилятор. В нашем случае gcc и ассемблер nasm.
2. Выбрать компоновщик. В нашем случае ld.
3. Выбрать утилиту для преобразования разделяемой библиотеки в файл формата .efi. В нашем случае objcopy.
4. Определиться с параметрами командной строки для gcc и nasm.
5. Определиться с параметрами командной строки для ld.
6. Определиться с параметрами командной строки для objcopy.
7. Последовательно вызвать компилятор, компоновщик и objcopy, передавая результат одной программы другой программе. (Вероятно, поэтому эти программы все вместе и называются toolchain).

Далее рассмотрим, какие параметры мы должны передать каждой из программ. Я привожу содержимое Makefile. Если вы незнакомы с Makefile-файлами и утилитой make, могу посоветовать хороший туториал по ним.

```
BUILD_DIR := ./build
SRC_DIRS := ./src
LIBDIR := /usr/lib
ARCH := x86_64

TARGET_NAME := asbootsap
```

```
OBJCOPY := objcopy
CC := gcc
LD := ld

FORMAT := efi-app-x86-64

SECTIONS := .text .sdata .data .dynamic .dynsym .rel .rela .reloc

CRT0 := $(shell find $(LIBDIR) -name crt0-efi-$(ARCH).o 2>/dev/null | tail -n1)

LDSCRIPT := $(shell find $(LIBDIR) -name elf_$(ARCH)_efi.lds 2>/dev/null | tail -n1)

# Note the single quotes around the * expressions. The shell will incorrectly expand th
# but we want to send the * directly to the find command.
SRCS := $(shell find $(SRC_DIRS) -name '*.c')
SRCS_ASM := $(shell find $(SRC_DIRS) -name '*.asm')

OBJS_ASM := $(patsubst ./src/%.asm,$(BUILD_DIR)/%.o,$(SRCS_ASM))
OBJS := $(patsubst ./src/%.c,$(BUILD_DIR)/%.o,$(SRCS))

.PRECIOUS: $(OBJS)
.PRECIOUS: $(OBJS_ASM)
.PRECIOUS: $(BUILD_DIR)/%.so

# String substitution (suffix version without %).
DEPS := $(OBJS:.o=.d)

INC_DIRS := $(shell find $(SRC_DIRS) -type d)
GNU_EFI_DIRS := /usr/include/efi /usr/include/efi/$(ARCH)
INC_DIRS := $(INC_DIRS) $(GNU_EFI_DIRS)

CPPFLAGS := $(addprefix -I,$(INC_DIRS)) \
    -MMD \
    -MP

CFLAGS := -fshort-wchar \
    -DGNU_EFI_USE_MS_ABI \
    -ffreestanding \
    -mno-red-zone \
    -Wall \
    -Werror \
    -fPIC \
    -O2
```

```
LDFLAGS=-T $(LDSCRIPT) \
    -Bsymbolic \
    -shared \
    -nostdlib \
    -znocombreloc \
    -L$(LIBDIR) \
    $(CRT0)

all: $(BUILD_DIR)/$(TARGET_NAME).efi
    mkdir -p ./build

deploy: all
    mkdir -p ./esp/efi/boot
    cp $(BUILD_DIR)/$(TARGET_NAME).efi ./esp/efi/boot/BOOTX64.EFI

start: all deploy
    ./start-qemu.sh

$(BUILD_DIR)/%.efi: $(BUILD_DIR)/%.so
    $(OBJCOPY) $(foreach sec,$(SECTIONS),-j $(sec)) --target=$(FORMAT) -S $< $@

$(BUILD_DIR)/%.so: $(OBJS) $(OBJS_ASM)
    $(LD) $(LDFLAGS) -o $@ $^ -lgnuefi -lefi

# Build step for C source
$(BUILD_DIR)/%.o: $(SRC_DIRS)/%.c
    mkdir -p $(dir $@)
    $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

$(BUILD_DIR)/%.o: $(SRC_DIRS)/%.asm
    mkdir -p $(dir $@)
    nasm -g -f elf64 -l $@.lst $< -o $@

.PHONY: clean
clean:
    rm -r $(BUILD_DIR)

-include $(DEPS)
```

Настройка компиляции:

1. Формирование списков исходных файлов на C и ассемблере:

```
SRCS := $(shell find $(SRC*DIRS) -name '*.c')
SRCS_ASM := $(shell find $(SRC_DIRS) -name '*.asm')
```

2. Формирование списков объектных файлов:

```
OBJS_ASM := $(patsubst ./src/%.asm,$(BUILD_DIR)/%.o,$(SRCS_ASM))
OBJS := $(patsubst ./src/%.c,$(BUILD_DIR)/%.o,$(SRCS))
```

3. Формирование списка директорий с заголовочными файлами:

```
INC_DIRS := $(shell find $(SRC_DIRS) -type d)
GNU_EFI_DIRS := /usr/include/efi /usr/include/efi/$(ARCH)
INC_DIRS := $(INC_DIRS) $(GNU_EFI_DIRS)
```

4. Формирование списка параметров для препроцессора C:

```
CPPFLAGS := $(addprefix -I,$(INC_DIRS)) \
-MMD \
-MP
```

5. Формирование списка параметров для компилятора C:

```
CFLAGS := -fshort-wchar \
-DGNU_EFI_USE_MS_ABI \
-ffreestanding \
-mno-red-zone \
-Wall \
-Werror \
-fPIC \
-O2
```

6. Правила для компиляции файлов C и ассемблера:

```
$(BUILD_DIR)/%.o: $(SRC_DIRS)/%.c
   $(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

$(BUILD_DIR)/%.o: $(SRC_DIRS)/%.asm
     nasm -g -f elf64 -l $@.lst $< -o $@
```

Настройка компоновщика:

1. Расположение библиотечных файлов, кода начальной загрузки для UEFI-приложения и пользовательского скрипта компоновки:

```
LIBDIR := /usr/lib
CRT0 := $(shell find $(LIBDIR) -name crt0-efi-$(ARCH).o 2>/dev/null | tail -n1)
LDSCRIPT := $(shell find $(LIBDIR) -name elf_$(ARCH)\_efi.lds 2>/dev/null | tail -
```

2. Формирование списка параметров для компоновщика:

```
LDFLAGS=-T $(LDSCRIPT) \
-Bsymbolic \
-shared \
-nostdlib \
-znocombreloc \
-L$(LIBDIR) \
$(CRT0)
```

3. Правило для компоновщика:

```
$(BUILD_DIR)/%.so: $(OBJS) $(OBJS_ASM)
    $(LD) $(LDFLAGS) -o $@ $^ -lgnuefi -lefi
```

Настройка objcopy

1. Выбор формата исполняемого файла. В нашем случае это должен быть EFI Image (PE32+/COFF)

```
FORMAT := efi-app-x86-64
```

2. Формирование списка секций, которые нужно скопировать в исполняемый файл:

```
SECTIONS := .text .sdata .data .dynamic .dynsym .rel .rela .reloc
```

3. Правило для objcopy:

```
$(OBJCOPY) $(foreach sec,$(SECTIONS),-j $(sec)) --target=$(FORMAT) -S $< $@
```

Выводы

К сожалению, полностью передать те ощущения, которые получаешь, когда ты видишь, как твой код загружает Linux, в статье невозможно, но я надеюсь, что вы прошли шаги и у вас получилось запустить свой загрузчик. Ну или хотя бы откомпилировали мой, и может, попробовали внести небольшие изменения. Загрузчик обладает рядом недостатков:

- я упростил обработку заголовка bzImage файла,
- обработка ошибок минимальная,
- работу с памятью я упростил тем, что в основном используется программный стек, а не куча. Но вместе с тем он целостно и наглядно иллюстрирует принципы написания загрузчика, способен загрузить современные ядра Linux и может послужить импульсом для написания более сложного и более пригодного для реальной эксплуатации загрузчика.

Теги: ruvds_статьи, linux, uefi, загрузка ос, загрузчик, системное программирование, gcc

Хабы: Блог компании RUVDS.com, UEFI, Программирование, Разработка под Linux, Системное программирование

Электропочта

202 Карма 38 Рейтинг

@artyomsoft

Пользователь

Подписаться

Комментарии 19

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ

Exosphere
15 часов назад

Ещё 10 ошибок авторов Хабра

11 мин 3K

+91 33 63

vital_pavlenko
21 час назад

Больше нет входа в IT. Только выход

2 мин 74K

+47 76 311

duran-duran
21 час назад

Трамплин в интернет: как мы ускорили запуск Яндекс Браузера

6 мин 3.2K

+42 9 29

Nickmob
16 часов назад

Введение в Angie: краткая история и отличия от Nginx

Простой 7 мин 2.7K

Ретроспектива

+37 28 6

Myskat_90
20 часов назад

Распределённый инференс и шардирование LLM. Часть 1: настройка GPU, проброс в Proxmox и настройка Kubernetes

Сложный 14 мин 1.7K

Туториал

+33 46 0

slava_rumin
15 часов назад

Мое производство электрощитов приносит 40 млн в год. Спасибо нейросетям и СССР за конструкторскую школу

Простой 14 мин 20K

Интервью

+28 40 49

AlexeyNadezhin
22 часа назад

Важное обновление BatteryTest 2

Простой 3 мин 2.4K

+27 36 9