Rozruch systemu LINUX

Wojciech Jaworski

Bartosz Miłosierny

Proces bootowania - wstęp

- bootowaniem (ang. *booting*) nazywamy proces *bootstrappingu* prowadzący do uruchomienia systemu operacyjnego po włączeniu komputera
- słowo *bootstrapping* jest nawiązaniem do niemieckiej legendzie o baronie Munchausenie, który po wpadnięciu do bagna był w stanie wyciągnąć z niego sam siebie ciągnąc się za włosy (w późniejszych wersjach tej historii hrabia używa rzemieni ze swoich butów – ang. *boot straps* – do wyciągnięcia się z morza – z czego narodził się termin *bootstrapping*)
- we wszelkim kontekście informatycznym słowo *bootstrapping* odnosi się do sytuacji, w której mały system uruchamia duży system

Proces bootowania – wstęp – c.d.

- sekwencja bootowania – to ciąg operacji, które muszą zostać wykonane przez komputer od momentu uruchomienia do momentu załadowania systemu operacyjnego
- wszystkie komputery (jako sprzęt) potrafią uruchamiać tylko te programy, które znajdują się w głównej pamięci – a większość programów (łącznie z systemami operacyjnymi) jest przechowywana na rozmaitych nośnikach
- zaraz po uruchomieniu komputer nie ma możliwości posiadanych przez OS – nie może ładować programów z dysku do pamięci
- prowadzi to do pozornie nierozwiązywalnego paradoksu

Proces bootowania – bootloadery

- rozwiązaniem są malutkie programy zwane *bootloaderami* (programami rozruchowymi), które idealnie realizują ideę boostrappingu
- bootloader nie posiada pełnej funkcjonalności systemu operacyjnego, ale potrafi załadować taką jego część, która pozwoli na jego całkowite uruchomienie
- program rozruchowy oraz cały system operacyjny może być pobierany także z innych urządzeń takich jak stacja dyskietek, napęd CDROM, dyski USB a nawet spoza komputera - z serwera w sieci lokalnej
- bootloader (a w zasadzie jego pierwszy etap) jest ładowany przez BIOS

Co się dzieje po włączeniu przycisku POWER? - BIOS

- kontrolę nad sprzętem przejmuje BIOS (ang. *Basic Input/Output System*)
- BIOS jest programem zapisanym na stałe w pamięci ROM i jest wykonywany przy każdym włączeniu komputera
- po włączeniu komputera BIOS dekompresuje swój kod z pamięci flash i ładuje się do pamięci RAM i stamtąd rozpoczyna swoje działanie
- opcje użytkownika dla programu BIOSu zapisywane są w pamięci CMOS (ang. *complementary metal-oxide-semiconductor*), której trwałość jest podtrzymywana przez niezależne źródło prądu (baterię)
- kod źródłowy dla BIOSu pochodzący z 80x86 jest dostępny w *IBM Technical Reference Manual*)

Co się dzieje po włączeniu przycisku POWER? – BIOS c.d.

- BIOS jest czasami nazywany *firmware'm* ponieważ jest integralną częścią sprzętu i jego wersja jest charakterystyczna dla produktów konkretnych firm (największe z nich to: American Megatrends Inc. Phoenix Technologies czy Award Software International)
- BIOS jest przechowywany na nośnikach typu EEPROM czy pamięć flash, co umożliwia jego aktualizację
- ponieważ aktualizacja BIOSu jest czynnością niebezpieczną wprowadzono kilka zabezpieczeń:
  - *dual bios* – oryginalny program BIOSu jest nienaruszony w postaci backupu
  - „*boot block*" – nienadpisywalna część BIOSu, która jest uruchamiana w pierwszej kolejności i testuje dalszą część kodu - np. sprawdza sumy kontrolne

Co się dzieje po włączeniu przycisku POWER? - POST

- BIOS zapewnia dostęp do fundamentalnych komponentów komputera
- jego funkcją jest także testowanie sprawności tych komponentów – zajmuje się tym procedura POST (Power On Self Test)
- główne funkcje BIOSU podczas wywoływania procedur POST to:
  - sprawdzenie integralności kodu programu BIOS
  - ustalenie powodu, dla którego POST zostało uruchomione
  - znalezienie, ustalenie rozmiaru i sprawdzenie głównej pamięci
  - znalezienie, zainicjalizowanie i skatalogowanie wszystkich magistrali
  - dostarczenie interfejsu do konfiguracji systemu (BIOS Setup)
  - zidentyfikowanie urządzeń zdolnych do bootowania

BIOS – wzloty i upadki

- w prostych systemach operacyjnych takich jak DOS – BIOS pokrywał prawie wszystkie operacje wejścia/wyjścia oraz bezpośredni dostęp do sprzętu
- wraz z rozwojem bardziej skomplikowanych systemów (Windows, Linux) rola BIOSu został zredukowana tylko do początkowej inicjalizacji i testowania sprzętu
- jednak w ostatnich latach, wraz z pojawieniem się technologii ACPI (*Advanced Configuration and Power Interface*) BIOS przejął cześć zaawansowanych funkcji takich jak:
  - zarządzanie energią
  - *hotplug* devices
  - kontrola termiczna

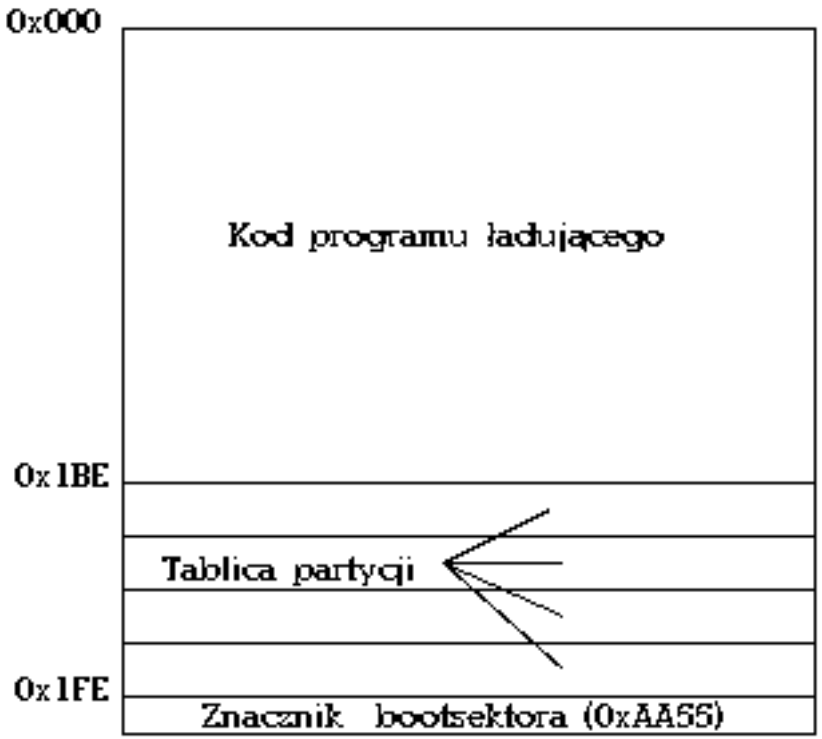
Proces bootowania

- po pomyślnym wykonaniu procedur testujących BIOS próbuje załadować pliki bootujące systemu operacyjnego
- w tym celu identyfikuje wszelkie napędy, które są zdolne do bootowania i sprawdza je w kolejności zdefiniowanej przez użytkownika
- możliwości BIOSu dotyczące napędów kończą się na wiedzy jak załadować pierwszy 512 bajtowy sektor z napędu
- BIOS ładuje ze stosownego nośnika pierwsze 512 bajtów do pamięci głównej pod adres 0000:7C00
- 2 ostatnie bajty z 512 bajtowego bloku muszą w przypadku programów bootujących mieć wartość 0x55AA (kolejność bajtów jest na dysku odwrócona) – po napotkaniu tego znacznika, BIOS skacze do adresu 0000:7C00

Struktura sektora bootującego

- bajty: 0..445 (446 bajtów) kod wykonywalny programu ładującego
- bajty 446..509 (64 bajty) tablica partycji (4 wpisy po 16 bajtów każdy – patrz następny slajd nr 11)
- bajty 510 i 511 (2 bajty) – znacznik stale równy: 0xAA55

Struktura Master Boot Recordu



Struktura tablicy partycji

```
// plik: include/linux/genhd.h
struct partition {
        unsigned char boot_ind;          /* 0x80 - aktywna */
        unsigned char head;              /* ścieżka początkowa */
        unsigned char sector;            /* sektor początkowy */
        unsigned char cyl;               /* cylinder początkowy */
        unsigned char sys_ind;           /* Typ partycji */
        unsigned char end_head;          /* ścieżka końcowa */
        unsigned char end_sector;        /* sektor końcowy */
        unsigned char end_cyl;           /* cylinder końcowy */
        unsigned int start_sect;         /* sektor początkowy partycji */
        unsigned int nr_sects;           /* liczba sektorów wchodzących
                                            w skład partycji */
} //razem 16 bajtów
```

Kilka informacji o tablicy partycji

- dysk twardy może posiadać do 4 tablicy podstawowych
- zamiast dowolnej z nich może wystąpić partycja rozszerzona zawierająca do 4 partycji zwanych logicznymi
- struktura dysku twardego jest jednoznacznie opisana za pomocą tablicy partycji
- tablica znajdująca się w MBR opisuje partycje podstawowe i rozszerzone znajdujące się bezpośrednio na dysku twardym
- w pierwszym sektorze dowolnej partycji rozszerzonej znajdzie się tablica partycji opisująca partycje logiczne umiejscowione na tej partycji rozszerzonej

Bootstrapping – ciąg dalszy

- w pierwszym sektorze każdej partycji jest zarezerwowane miejsce, w którym jest umieszczony kolejny program ładujący, wczytywany i uruchamiany przez ten z MBR
- zadania programu z MBR często ograniczają się do załadowania tego programu, który znajduje się na partycji oflagowanej znacznikiem: *active flag* (pierwsze pole struktury partition)
- zaawansowane bootloadery (np. LILO), które można umieścić w MBR nie korzystają z tej flagi

Jądro systemu Linux

- obraz jądra powstaje w wyniku kompilacji plików ze źródłami
- współczesne jądra są zwykle większe niż 512KB – dlatego są kompresowane (big kernel bzImage; zImage)
- podczas wczesnego bootowania, jądro ma do dyspozycji tylko niesławne 640KB pamięci – działa w trybie rzeczywistym procesora
- dlatego potrzebne są „tricki” z przerzucaniem części jądra do różnych obszarów pamięci przed włączeniem trybu chronionego
- na samym początku bardzo ważne są części jądra:
  - *bootsect.S* - malutki program, który jest zapisany w bootsektorze; jego zadaniem jest wczytanie i wystartowanie głównej części jądra
  - *setup.S* – odpowiada za pobraniu z BIOSu danych systemowych i umieszczenie ich w odpowiednich miejscach pamięci (dysk, pamięć i inne parametry)

Ładowanie jądra linuksa - scenariusz dla zImage (<512KB)

- bootsector (*bootsector.S* uruchamiany w pamięci w miejscu 0x7c00 w trybie rzeczywistym) przesuwa swój kod w miejsce 0x90000 i wczytuje – korzystając z przerwań BIOSu – sektory znajdujące się tuż za nim
- reszta kernela jest ładowana pod adres 0x10000 (0x90000 – 0x10000 = 512KB) wypełniając maksymalnie do pół megabajta
- kod załadowany w miejscu 0x90200 (0x90000 + 512 bajtów na bootsector.S) – zdefiniowany w pliku *setup.S* zajmuje się inicjalizacją sprzętu i pozwala na zmianę trybu wyświetlania (*video.S*)
- później cały kernel jest przesuwany z adresu (0x10000 = 64K) pod adres (0x1000 = 4K) i w ten sposób nadpisuje dane BIOSu trzymane w pamięci RAM – wywołania BIOS-owe nie mogą już mieć miejsca
- w tym punkcie *setup.S* wchodzi w tryb chroniony procesora i skacze pod adres 0x1000, gdzie umieszczony jest kernel. W trybie chronionym cała pamięć jest już dostępna i system może zacząć się uruchamiać (wywołanie funkcji: *start_kernel()*)

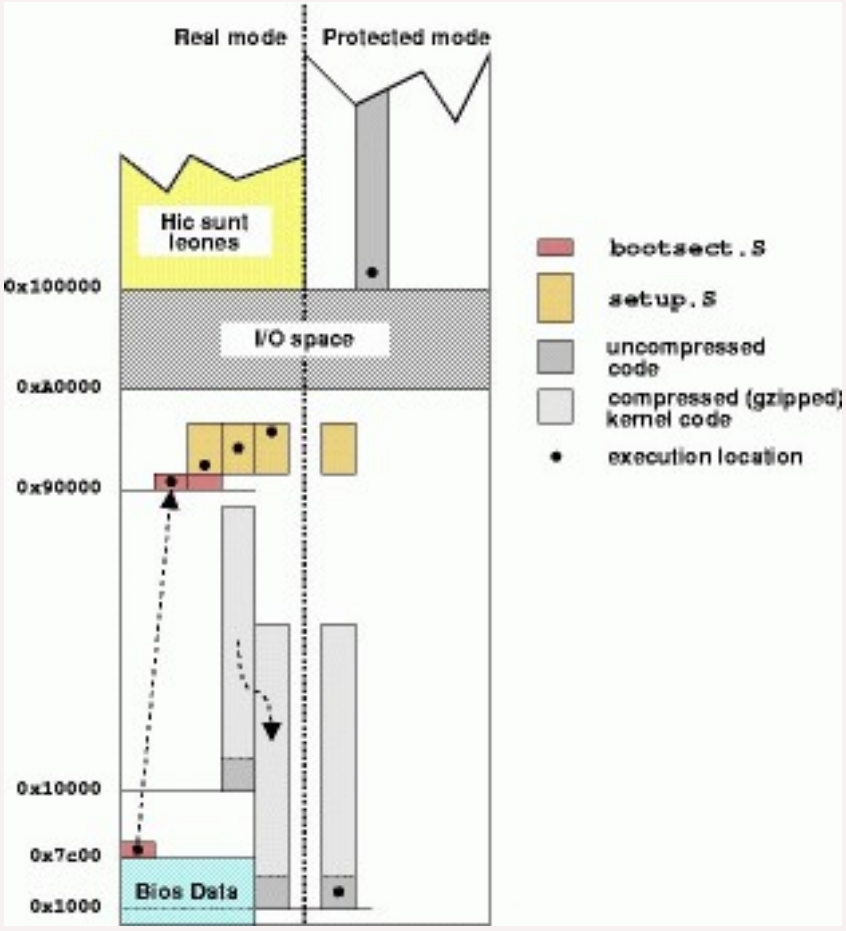
Kompresowane jądra

- powyższy scenariusz miał miejsce w czasach, kiedy kernel był mały, na tyle mały, by zmieścić się w 512K
- z czasem do kernela włączano rozmaite ficzury i kernel rozrósł się do gigantycznych rozmiarów (dla wielu wyczynem jest zmieszczenie własnoręcznie skonfigurowanego jądra na dyskietce 1,44M)
- kod większy niż pół megabajta nie może być przesuwany pod adres 0x1000 – nie zmieści się w przydzielonym fragmencie pamięci
- dlatego – w tym przypadku - w miejscu 0x1000 nie leży jądro, tylko część „gunzip” programu **gzip**

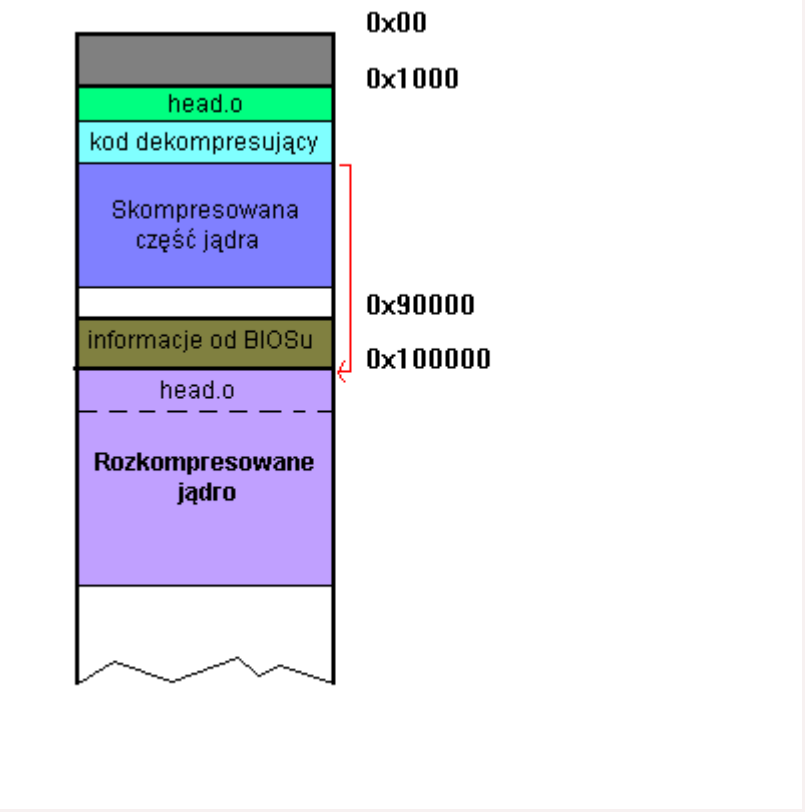
Kompresowane jądra - 2

- *head.S* jest umieszczany pod adresem 0x1000
- jego zadaniem jest rozzipowanie kernela – wywołuje on funkcję *decompress_kernel* (compressed/misc.c), która z kolei woła funkcję *inflate*
- output tej funkcji jest pisany do pamięci począwszy od adresu 0x100000 (1M)
- dostęp do pamięci powyżej 1M jest zapewniony, ponieważ pracujemy w trybie chronionym
- po dekompresji *head.S* skacze do aktualnego początku jądra
- proces bootowania jest teraz zakończony i kod, który znajduje się w 0x100000 (ten sam kod, który był w 0x1000 w nieskompresowanych jądrach) zajmuje się inicjalizacją i wywołaniem funkcji *start_kernel()*

Przemieszczanie jądra w pamięci podczas bootowania



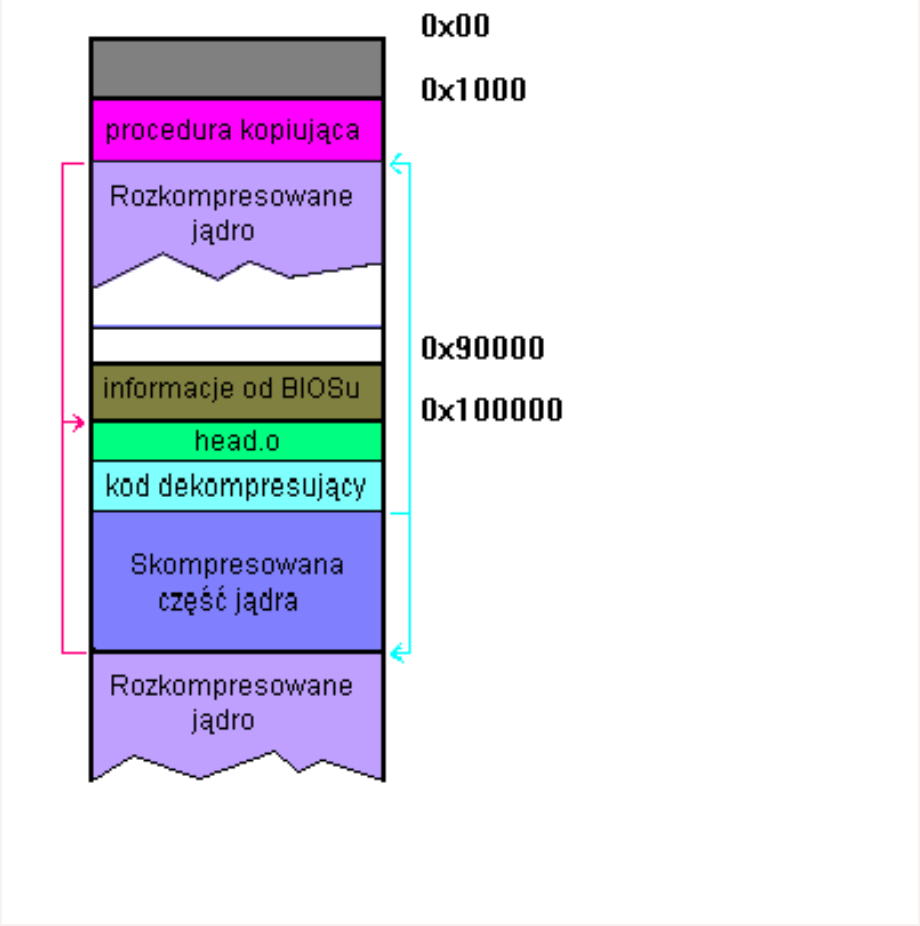
Dekompresja zwykłego jądra



Kompresowane jądra – metoda bzImage

- począwszy od jądra 1.3.73 dostarczono nowe narzędzie – *make bzImage*
- ten rodzaj jądra ładuje się w trochę inny sposób – zachodzą pewne zmiany:
  - kiedy system jest ładowany pod adres 0x10000, zostaje wywołany pewien pomocniczy kod po odczytaniu każdych 64K danych. Ta pomocnicza funkcja przenosi blok danych do pamięci wysokiej korzystając z funkcji biosowych
  - *setup.S* nie przesuwa systemu z powrotem do 0x1000 ale, po wkroczeniu w tryb chroniony skacze bezpośrednio do 1M, gdzie leżą dane przeniesione przez BIOS w poprzednim kroku
  - dekompresor, którego kod znajduje się pod adresem 0x100000 (1M) rozpakowuje kernel do niskiej pamięci, dopóki nie zostanie wyczerpana, a potem do pamięci wysokiej, zaraz za skompresowanym kodem
  - te dwa kawałki kodu są łączone i zapisywane pod adres (0x100000) – aby wykonać to zadanie prawidłowo potrzeba kilka przerzutów kodu w pamięci

Dekompresja dużego jądra



Informacje dodatkowe

- zadania *setup.S:*
  - Sprawdzenie wersji loadera, poprawności załadowania i rodzaju jądra
  - Rozmiar pamięci (BIOS)
  - Rozpoznanie i inicjalizacja karty graficznej
  - Ilość i parametry dysków podłączonych do pierwszego kontrolera
  - Obecność architektury MCA
  - Mysz PS/2
  - BIOS APM (ACPI Power Management)

Informacje dodatkowe - 2

- przełączenie w tryb chroniony:
  - Wyzerowanie lokalnej tablicy deskryptorów (LDT)
  - Ustawienie w globalnej tablicy deskryptorów (GDT) wskaźników na 4 GB segmenty kodu i danych jądra, zaczynające się od 0
  - Wyłączenie przerwań
  - Włączenie linii adresowej A20
  - Reset koprocesora
  - Przesunięcie wektorów przerwań programowych pod 0x20
  - Załadowanie słowa stanu procesora z ustawionym bitem trybu chronionego

Program rozruchowy

- Zazwyczaj składa się z kilku mniejszych programów:
  1. Boot Sector Program – kluczowy punkt procesu bootowania
  2. Second Stage Program (i ewentualnie kolejne fazy)
  3. Dodatkowe narzędzia, w tym program instalujący bootloader na dysku

Boot Sector Program - zadania

- Załadowanie innego bootloadera (częste, gdy mamy kilka systemów operacyjnych)
- Odnalezienie na dysku Second Stage Program, załadowanie go i uruchomienie
- Załadowanie jądra systemu już na tym etapie (np. bootloader systemu DOS)

Second Stage Boot Program

- Wykonuje większość zadań oczekiwanych od programu bootującego
- Brak ograniczeń na wielkość
- Zapewnia interfejs użytkownika pozwalający na wybór systemu
- Ładuje do pamięci operacyjnej jądro systemu
- Załadowanie innego bootloadera (chain loading)

Popularne bootloadery Linuksa

- Platforma i386
  - LILO: http://lilo.go.dyndns.org/
  - GRUB: http://www.gnu.org/software/grub/
  - GRUB 2: http://www.gnu.org/software/grub/grub-2.en.html
  - SYSLINUX: http://syslinux.zytor.com/
  - GUJIN: http://gujin.sourceforge.net/
- Inne platformy
  - GRUB 2
  - SILO: http://www.sparc-boot.org/
  - Quik Bootloader: http://www.penguinppc.org/bootloaders/quik/
  - BootX: http://www.penguinppc.org/bootloaders/bootx/

LILO - LInux LOader

- Pierwszy zaawansowany bootloader dla systemu Linux
- Obecnie stracił znaczenie na rzecz programu GRUB
- Aktualna wersja: 22.7.1
- http://lilo.go.dyndns.org/

LILO – możliwości

- Start Linuksa z bieżącego dysku, innego dysku twardego, dyskietki
- Wybór jądra, które ma zostać załadowane
- Wybór innego systemu operacyjnego
- Niezależność od systemu plików – LILO wczytuje obraz jądra na podstawie numeru sektora dysku, nie korzystając ze struktury systemu plików
- Po każdej zmianie konfiguracji należy ponownie zainstalować LILO

Pakiet LILO – 2 części

- Pliki wykorzystywane do instalacji właściwego bootloadera:
  - /sbin/lilo
  - /etc/lilo.conf
- Pliki wykorzystywane do rozruchu systemu:
  - /boot/boot.b – kod wykonywalny bootloadera
  - /boot/map – numery sektorów z obrazami jąder
  - /boot/chain.b – używane do ładowania innych systemów

lilo.conf - przykład

```
boot=/dev/hda                       # lokalizacja programu ładującego
map=/boot/map                       # numery niezbędnych sektorów
install=/boot/boot.b
prompt
timeout=50
default=linux
image=/boot/vmlinuz-2.6.12          # image dla obrazów jąder
        label=linux
        read-only
        root=/dev/hda1
other=/dev/hda1
        label=msdos                 # other przy chainloadingu
        table=/dev/hda
        loader=/boot/chain.b
```

LILO Boot Sector Program

- Ładowanie systemu rozpoczyna się od wczytania sektora ładującego zapisanego w MBR lub w sektorze startowym partycji
- Jest to kopia pierwszych 512 bajtów pliku boot.b
- Zawiera assemblerowy kod Primary Boot Loader oraz strukturę:

```
typedef struct {
                unsigned char cli;
                unsigned char jmp0, jmp1;
                unsigned char stage;
                unsigned short code_length;
                char signature[4];
                unsigned short version;
                unsigned int map_stamp;
                unsigned int raid_offset;
                unsigned int timestamp;
                unsigned int map_serial_no;
                unsigned short prompt;
                SECTOR_ADR secondary; /* Sektor drugorzędnego programu ladujacego
           */
} BOOT_PARAMS_1; /* first stage boot loader */
```

LILO Second Stage Program

- Podstawowy program ładujący odczytuje strukturę BOOT_PARAMS_1 inicjuje stos i ładuje drugorzędny program ładujący.
- Drugorzędny programu ładującego zajmuje pozostałą część pliku boot.b. Jego rozmiar nie może przekraczać 8 sektorów.
- Wczytuje tablicę deskryptorów (IMAGE_DESCR) obrazów ładowalnych (plików jąder lub struktury zawierającej kopię pośredniego programu ładującego i pierwszego sektora partycji przy chainloadingu). Tablica ta może mieć do 16 wpisów, jest zapisana w /boot.map)
- LILO inicjuje linię poleceń i menu (pierwszy sektor /boot/map)
- Przetwarzana jest linia poleceń otrzymana przez użytkownika i odszukany zostaje w tablicy deskryptorów odpowiedni obraz
- Uruchomiony zostaje kod setup.o jądra Linuksa

GRUB - GRand Unified Bootloader

- Najpopularniejszy obecnie bootloader Linuksa
- http://www.gnu.org/software/grub/
- Stworzony początkowo przez Ericha Stefana Boleyna
- Ostatnia wersja: 0.97
- Prace nad projektem zostały zakończone, obecnie rozwijany jest program GRUB 2

GRUB - możliwości

- Multibootloader (Linux, FreeBSD, GNU Hurd…), chainloading
- Rozpoznaje wiele formatów wykonywalnych
- Potrafi odczytywać systemy plików (ext2, ext3, ReiserFS, FAT, JFS…)
- Ładuje dynamicznie konfigurację
- Uzyskuje dostęp do całego dysku i pamięci RAM
- Udostępnia shellopodobną powłokę (ładowanie jądra, ramdysku, listowanie plików, autouzupełnianie)
- Bogatsze możliwości menu graficznego
- Pozwala na bootowanie przez sieć
- Pozwala na ochronę hasłem

GRUB - instalacja

- Skrypt *grub-install*

```
grub-install install_device [--force-lba] [--root-
directory=dir]
```

- Konsola *grub*

```
grub> root install_device
grub> setup [--force-lba] [--stage2=os_stage2_file]
       [--prefix=dir] install_device [image_device]
```

grub.conf / menu.lst

```
default=2
timeout=5
vga=789
splashimage=(hd0,7)/grub/splash.xpm.gz
hiddenmenu
title Fedora Core (2.6.12-1.1456_FC4)
        root (hd0,7)
        kernel /vmlinuz-2.6.12-1.1456_FC4 ro root=/dev/hda9 rhgb quiet
        initrd /initrd-2.6.12-1.1456_FC4.img
title Fedora Core (2.6.11-1.1369_FC4)
        root (hd0,7)
        kernel /vmlinuz-2.6.11-1.1369_FC4 ro root=/dev/hda9 rhgb quiet
        initrd /initrd-2.6.11-1.1369_FC4.img
title Windows
        rootnoverify (hd0,0)
        chainloader +1
```

GRUB - składniki programu rozruchowego:

- Pliki umieszczone najczęsciej w /boot/grub
  - stage1 – Boot Sector Program
  - [fstype]_stage1_5 –ładuje obsługę systemu plików
  - stage2 - Second Stage Program

stage1

- Zapisany w MBR lub sektorze startowym partycji
- 512 bajtów – kod assemblerowy, tablica partycji
- Na tym etapie GRUB nie rozumie jeszcze systemu plików, uruchamia kolejny plik na podstawie adresu sektora
- Jest odpowiedzialny wyłącznie za załadowanie do pamięci pierwszego sektora stage2 lub stage1_5
- Budowa:

```
0x3E     The version number (not GRUB's, but the installation mechanism's).
0x40     The boot drive. If it is 0xFF, use a drive passed by BIOS.
0x41     The flag for if forcing LBA.
0x42     The starting address of Stage 2.
0x44     The first sector of Stage 2.
0x48     The starting segment of Stage 2.
0x1FE    The signature (0xAA55).
```

stage1_5 i stage2

- Pliki: e2fs_stage1_5, fat_stage1_5, iso9660_stage1_5,…
- Ładują obsługę systemu plików
- GRUB stosuje następujące oznaczenia dysków:
  - (fd0) – dyskietka
  - (hd0,1) – druga partycja pierwszego dysku
  - (hd0,0)/boot/vmlinuz – dostęp do plików
- Stage2 ładuje resztę swojego kodu przeprowadza pozostałe czynności rozruchu systemu
- Stage1_5 i stage2 są napisane w C

GRUB 2 - cele

- Pełna zgodność z Multiboot Specification
  http://www.gnu.org/software/grub/manual/multiboot/
- Zbudowanie przenośnego obrazu programu rozruchowego (połączenie stage1.5 i stage2)
- Cross-platform installation
- Oddzielenie kodu specyficznego dla Intela od generycznego i wsparcie dla innych platform
- Object-oriented framework dla systemu plików, urządzeń
- Wsparcie dla wersji językowych
- TODO List: http://grub.enbug.org/TodoList

GRUB 2 - postęp prac

- Wspierane architektury

  PC (i386), Mac (powerpc), Pegasos II (powerpc), Sparc v9 (Sun UltraSparc) - under developement

- Wspierane systemy plików

  ext2, fat (+long filenames), ufs, minix, iso9660, jfs, hfs, affs, sfs, xfs

- Tablice partycji

  Standard PC and extended partitions

  BSD partitions

  Macintosh partitions

  Amiga style partitions (RDB)

  Sun partitions

GRUB 2 - postęp prac (cd.)

- Wyświetlanie
  - PC: textmode, VESA framebuffer (work in progress)
  - PPC & UltraSparc: ANSI (using Open Firmware)
- Zaimplementowane komendy:

  boot cat cmp configfile halt help insmod loopback ls lsmod reboot rescue rmmod search set terminal test unset
- Inne cechy:
  - Ulepszone zarządzanie pamięcią
  - Emulator GRUB-a (grub-emu)
  - Używanie zmiennych
  - Ładowanie modułów
  - Historia poleceń i autouzupełnianie w linii komend

GRUB 2 – jak przetestować

- cvs -z3 -d:ext:anoncvs@savannah.gnu.org:/cvsroot/grub co grub2
- Ostatnia wersja: 1.92 dostępna przez ftp:

  ftp://alpha.gnu.org/gnu/grub/
- W najbliższych wersjach ma się pojawić obsługa sieci, ReiserFS, graficzne menu, manual(!)
- *grub-install* – skrypt shellowy, najprostsza metoda instalacji GRUB 2
- Programy wykorzystywane przez skrypt:
  - *grub-mkdevicemap*
  - *grub-probefs*
  - *grub-mkimage*
  - *grub-setup*

GRUB 2 a GRUB

- Pliki bootloadera:

```
stage 1 = boot.img
stage 1.5 = diskboot.img + core.img + pc.mod + ext2.mod
stage 2 = normal.mod +_chain.mod
```

- Plik grub.cfg

```
timeout 10
title Linux
    linux (hd0,0)/vmlinuz-2.6.12-1.1456_FC4 root=/dev/hda1
    initrd (hd0,0)/initrd-2.6.12-1.1456_FC4.img
```

- Porównanie komend:

http://grub.enbug.org/CommandList