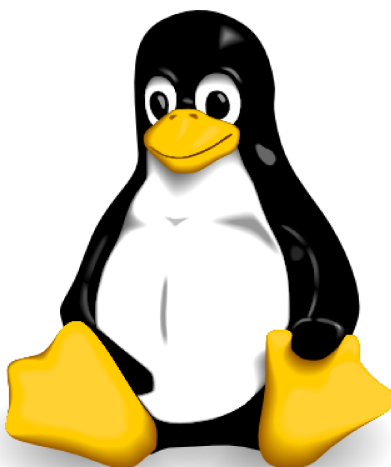
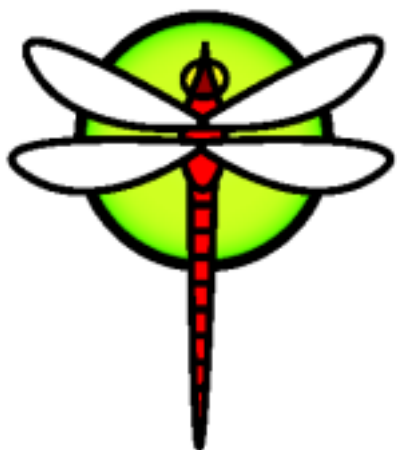




Aldis Berjoza

# Assembler programmēšana Un\*x vidē



# Apskatāmās Unix veidīgās operētājsistēmas

## BSD:

FreeBSD	<a href="http://www.freebsd.org">http://www.freebsd.org</a>
OpenBSD	<a href="http://openbsd.org">http://openbsd.org</a>
NetBSD	<a href="http://netbsd.org">http://netbsd.org</a>
DragonflyBSD	<a href="http://dragonflybsd.org">http://dragonflybsd.org</a>

## GNU/Linux:

Visi n-zin-cik distributīvi, Kodols	<a href="http://kernel.org/">http://kernel.org/</a>
----------------------------------------	-----------------------------------------------------

# Rīki

## Tradicionālie rīki

**as** jeb **gas** – GNU assembler  
**ld** – GNU linker  
**gdb** – GNU debugger  
**objdump**

## Kodola izsaukumu sekotāji

**ktrace** & **kdump** (uz BSD)  
**strace** (uz GNU/Linux)

## Citi assembleri:

**Fasm** – flat assembler  
**Nasm** – netwide assembler  
**Yasm**

## Citi debuggeri

**ald**  
**edb**

## Citi rīki

**procstat** (uz FreeBSD)  
**lsof** (uz Linux)

*EDB - pašlaik darbojas tikai uz Linux un, iespējams, uz OpenBSD, bet nākamā versija varētu darboties arī uz FreeBSD [tā cer EDB autors]*

# GAS – GNU Assembler

## Plusi:

- Parasti jau iekļauts operētājsistēmā (var saukties as vai gas)
- Spēj nolasīt C/C++ galvotnes (headers)
- Var atklūdot ar gdb
- Ļoti pārnēsājams

## Mīnusi

- Pēc noklusējuma izmanto AT&T sinteksi

*GAS failiem pieņemts izmantot .s paplašinājumu (file extension)*

<http://www.gnu.org/software/binutils/>

# GDB – GNU debugger

Plusi:

- Parasti iekļauts operētājsistēmā
- Labs atklūdotājs C/C++ programmām
- (runā, ka) darbojas, ja assemblera kods ir kompilēts ar GAS un ir atklūdošanas simboli (debugging symbols)
- Ļoti pārnēsājams

Mīnusi:

- Ja nav pieejams programmas kods – GDB nekam neder
- Nepierasts interfeiss
- Jauniem lietotājiem daudz grūtāks nekā Turbo Debugger un rīki uz Windows

<http://www.gnu.org/software/gdb/>

# FASM – Flat Assembler

Plusi:

- SSSO (Same Source Same Output) princips
- Pārnēsājamība
- Vienkārša sintakse
- Ļoti jaudīgi makrosi
- Multipass
- Nav jāraksta SHORT un LONG pie jmp
- Ļoti laba dokumentācija un forums

Mīnusi

- Atklūdošanas simboli (debugging symbols) nav savietojami ar GDB

<http://flatassembler.net/>

Minētie piemēri būs rakstīti FASM assembleram

# NASM – Netwide assembler

Plusi:

- Vienkārša sintakse
- Popularitāte

Mīnusi

- Jāraksta SHORT un LONG pie jmp

<http://www.nasm.us/>

YASM ir veidots uz NASM bāzes

<http://www.tortall.net/projects/yasm/>

# ALD – Assembly Level Debugger

Plusi:

- Darbojas ja nav programmas koda
- Vienīgais sakarīgais asm atklūdotājs, kas strādā gan uz GNU/Linux, gan BSD

Mīnusi

- Izskatās ka, vairākus gadus projekts ir pamests (v0.1.7 – 2004.g. 10. oktobris)
- Interfeiss apmēram tāds pats, kā GDB
- Kļūdas programmā

Un tomēr tas strādā... :)

<http://ald.sourceforge.net/>



# EDB

## Plusi:

- Vienkāršs interfeiss
- Viegli strādāt
- Tiek aktīvi izstrādāts

## Mīnusi:

- Pašlaik darbojas tikai uz GNU/Linux
- Grafiskais interfeiss

<http://www.codef00.com/projects.php#debugger>

# Un\*x programēšana

Tradicionāli Un\*x programmē C valodā – tādējādi nodrošinot lielu ātrdarbību un relatīvi vienkāršu koda pārnēsājamību starp dažādu procesoru arhitektūrām (IA-32, IA-64, Itanium, ARM, Sparc, PowerPC...).

Assembler parasti izmanto ļoti reti, specifisku uzdevumu risināšanai, kad piemēram ar C nevar uzrakstīt pietiekami mazu vai ātru kodu, piemēram MBR (Master Boot Record)

Tomēr assembler var izmantot, lai risinātu ļoti sarežģītus algoritmus, lielā ātrumā... tādēļ der zināt, kā darbojas Un\*x un kā programmēt assemblerā

# Sistēmas izsaukumi

Sistēmu var izsaukt:

- Izmantojot libc – standarta bibliotēku (labāka pārnēsājamība starp operētājsistēmām)
- Vēršoties pie sistēmas kodola pa tiešo (ātrāk)

Abos gadījumos informācija ir pieejama Un\*x pamācībās Otrajā sekcijā (man 2).

Otrās sekcijas informācija ir paredzēta sistēmas izsaukumu aprakstīšanai, ja izmanto libc rakstot C programmas. Tomēr šī informācija noder assembler programmētājiem jo satur informāciju par datu tiem, konstantēm utt.

# Sistēmas izsaukumi

Un\*x sistēmās (Linux, BSD) kodola izsaukšanai izmanto **0x80** pārtraukumu

**int 0x80**

Parametrus nodod:

- caur steku (BSD sistēmas)
- izmantojot reģistrus (Linux)

*BSD sistēmas arī atbalsta (ja ir attiecīgi konfigurētas un kompilētas) Linux izsaukumus (bet tas šajā stāstā netiks apskatīts)*

*Citas Un\*x sistēmas var darboties savādāk, piemēram Solaris izmanto int 0x91 un parametru nodošana atšķiras gan no Linux, gan no BSD sistēmām*

# BSD sistēmu kodola izsaukumi

EAX reģistrā ieraksta sistēmas izsaukuma numuru  
Parametrus nodod caur steku (no labās uz kreiso pusi)  
Stekā ievieto vienu tukšu argumentu (dummy arg)  
Kodolu izsauc ar 0x80 pārtraukumu

Pēc izsaukuma veikšanas steks nav mainījies, tas pašam ir jāattīra  
Ja radās kļūda, tad tiks uzstādīts CF un kļūdas kods ievietots EAX reģistrā  
Ja kļūda nav radusies sistēma atgriež vērtības EAX reģistrā

Steka attīrīšanai vislabāk izmantot LEA instrukciju, jo tā nemaina karodziņu reģistru (atšķirībā no ADD instrukcijas)

*Ļoti retos gadījumos vērtība tiek atgriezta EDX reģistrā (piem. SYS\_FORK)*

# HELLO WORLD uz BSD sistēmām FASM sintaksē

```
format ELF ; ELF programma

STDOUT      = 1 ; STDOUT faila deskriptors
EXIT_SUCCESS = 0
EXIT_FAILURE = 1
SYS_WRITE   = 4 ; Rakstīšanas funkcijas numurs
SYS_EXIT    = 1 ; Programmas beigšanas funkcijas numurs

section '.text' executable ; KODA SEGMENTS
public _start
_start: ; Programmas ieejas punkts
    push    msg_len ; izvadāmā teksta garums
    push    msg      ; rādītājs uz tekstu
    push    STDOUT    ; Faila deskriptors, kurā rakstīsim
    push    eax        ; tukšs arguments
    mov     eax,SYS_WRITE ; funkcija
    int     0x80        ; izsaucam kodolu
    lea     esp,[esp+4*4] ; Tīrām steku
    jc      some_error  ; varbūt bija kļūda?

    push    EXIT_SUCCESS ; programmas beigšanas statuss
exit:
    push    eax ; tukšs arguments
    mov     eax, SYS_EXIT ; Funkcija
    int     0x80 ; GAME OVER
some_error:
    push    EXIT_FAILURE ; programmas beigšanas statuss
    jmp     exit

section '.data' writeable ; DATU SEGMENTS
msg      db "Hello world",0xA ; teksts kuru vēlamies izvadīt
msg_len  = $-msg ; teksta garums
```

# Programmas kompilēšana

```
$ fasm hello.fasm hello.o  
flat assembler version 1.69.24 (16384 kilobytes memory)  
3 passes, 446 bytes.  
$ ld -s -o hello hello.o
```

FASM ideoloģija paredz, ka viss nepieciešamais programmas kompilēšanai ir jāieraksta failā, tādējādi nav jāizmanto nekādi parametri programmas kompilēšanai (KISS)

ld pēc noklusējuma vienmēr meklē programmas ieejas punktu ***\_start***, ja tāda nav, tad tiks izvadīts paziņojums par kļūdu (citu ieejas punktu var norādīt ar **-e** parametru)

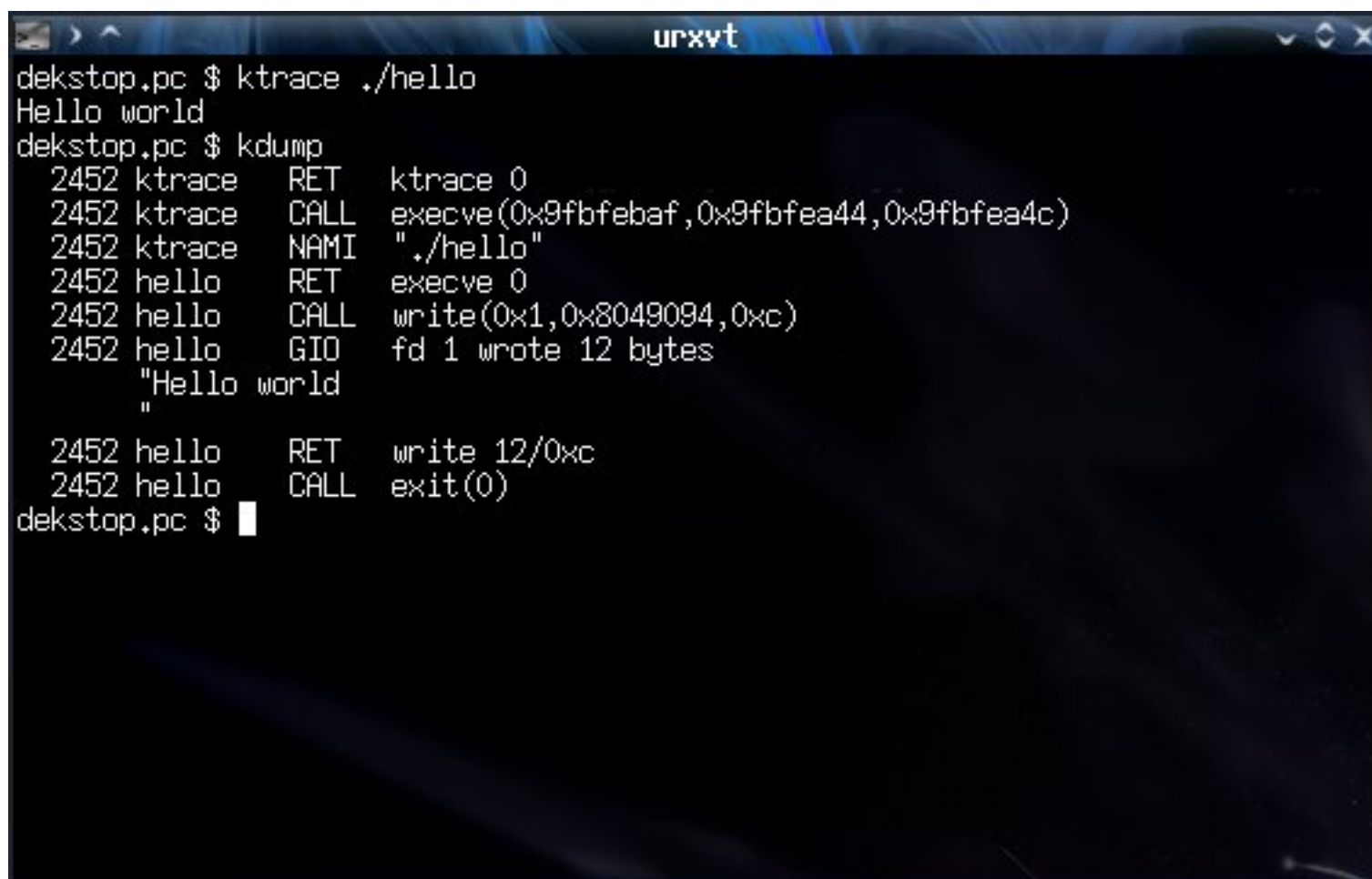
ld parametri:

- s** – strip symbolic debugging information
- o filename** – output filename
- e entry** – Entry point name

Sīkāka informācija [ld\(1\)](#)

# ktrace un kdump

ktrace un kdump ļauj ātri un vienkārši virspusēji ieskatīties, kas notiek programmā (pat, ja nav pieejams programmas kods), izanalizējot sistēmas kodola izsaukumus

A terminal window titled 'urxvt' showing the execution of 'ktrace ./hello' and 'kdump'. The output of 'kdump' shows system calls like execve, write, and exit for the process 'hello'.

```
dekstop.pc $ ktrace ./hello
Hello world
dekstop.pc $ kdump
2452 ktrace  RET   ktrace 0
2452 ktrace  CALL  execve(0x9fbfebfaf,0x9fbfea44,0x9fbfea4c)
2452 ktrace  NAMI  "./hello"
2452 hello   RET   execve 0
2452 hello   CALL  write(0x1,0x8049094,0xc)
2452 hello   GIO   fd 1 wrote 12 bytes
      "Hello world"
      "
2452 hello   RET   write 12/0xc
2452 hello   CALL  exit(0)
dekstop.pc $
```



# BSD kodola izsaukumu numuri

Informāciju par kodola izsaukumu numuriem un parametriem var atrast failā:

**[/usr/src/sys/kern/syscalls.master](#)**

Šo failu var apskatīties arī internetā:

- <http://fxr.watson.org/fxr/source/kern/syscalls.master?v=FREEBSD8>  
(FreeBSD 8)
- <http://fxr.watson.org/fxr/source/kern/syscalls.master?v=NETBSD5>  
(NetBSD 5)
- <http://fxr.watson.org/fxr/source/kern/syscalls.master?v=OPENBSD>  
(OpenBSD)
- <http://fxr.watson.org/fxr/source/kern/syscalls.master?v=DFBSD>  
(DragonflyBSD)

*Neskatoties uz BSD sistēmu kopējo vēsturi, kodola izsaukumu numuri atšķiras.*

Projekts BSD sistēmu FASM iekļaujamo failu izstrādei:

<http://hg.bsdroot.lv/pub/aldis/asm4BSD>

# Konstantes, kļūdu kodi un datu struktūras

Rokasgrāmatās (man pages) bieži tiek minētas dažādas konstantes, bet to vērtības nav dotas.

Konstanšu vērtības un datu struktūras jāmeklē pašam C galvotnēs.

Problēma ir arī tāda, ka reizēm vajag pārbaudīt, vai datu tips ir 32 bitu vai 64 bitu

*GAS assembleram šī problēma nav tik būtiska, jo tā failos var iekļaut C galvotnes*

# Linux kodola izsaukumi

EAX reģistrā ievieto funkcijas numuru

EBX, ECX, EDX, ESI, EDI, EDP – reģistros (minētajā secībā) ievieto parametru vērtības

Izsauc 0x80 pārtraukumu

Funkcijas rezultātā EAX tiek ievietota atgrieztā vērtība, vai kļūdas kods

Ļoti labs, precīzs un detalizēts avots par Linux kodola izsaukumiem:

<http://sourceforge.net/projects/lscr/files/>

Ļoti retos gadījumos, ja nepietiek reģistru visu parametru nodošanai, tad EBX reģistrā ievieto rādītāju (adresi) uz vietu atmiņā, kur ir izvietoti parametri

# HELLO WORLD uz Linux FASM sintaksē

```
format ELF                      ; ELF programma

STDOUT      = 1                ; STDOUT faila deskriptors
EXIT_SUCCESS = 0
SYS_WRITE   = 4                ; Rakstīšanas funkcijas numurs
SYS_EXIT    = 1                ; Programmas beigšanas funkcijas numurs

section '.text' executable      ; KODA SEGMENTS
public _start
_start:                          ; Programmas ieejas punkts
    mov     edx,msg_len          ; izvadāmā teksta garums
    mov     ecx,msg              ; rādītājs uz tekstu
    mov     ebx,STDOUT           ; Faila deskriptors, kurā rakstīsim
    mov     eax,SYS_WRITE        ; funkcija
    int     0x80                 ; izsaucam kodolu

    mov     ebx,EXIT_SUCCESS      ; programmas beigšanas statuss
exit:
    mov     eax,SYS_EXIT          ; Funkcija
    int     0x80                 ; GAME OVER

section '.data' writeable       ; DATU SEGMENTS
msg         db    "Hello world",0xA ; teksts kuru vēlamies izvadīt
msg_len     =     $-msg           ; teksta garums
```

*Šajā piemērā es nepārbaudu kļūdas*

# libc izmantošana

Izmantojot libc standarta bibliotēku var ievērojami palielināt programmas pārnēsājamību starp dažādām operētājsistēmām, jo nav jāuztraucas par veidu, kā izsaukt sistēmas kodolu.

Bonusā, nav jāizplata programmas kods, bet var izplatīt bināru moduli (gadījumā, ja programmas kods ir komerciāls noslēpums [Lai gan, kodu nevar noslēpt, principā])

Lai izmantotu libc assembler kodā jāimportē libc funkcijas. Piemēram

```
extrn printf  
extrn exit
```

Libc funkcijas (kā jau visas C funkcijas) parametrus saņem caur steku.  
Parametri tiek nodoti no labās puses uz kreiso.  
Pēc libc funkciju izsaukšanas jātīra steks pašam (lea ...)

Funkciju vērtības un kļūdas tiek atgrieztas EAX reģistrā.  
Lai zināt, kā noteikt kļūdas, jālasa katras funkcijas apraksts (man).

Protams izmantojot libc, programma būs lielāka un lēnāka

# libc izmantošana

**ld** saīšu redaktora vietā gan labāk izmantot **gcc** kompilatoru. Jo **ld** var nākties nodot daudz dažādu parametru, kurus grūti paredzēt. Uz vairuma Un\*x sistēmu, ja **gcc** ir iekļauts sistēmā to parasti sauc **cc**

```
$ fasm prog.fasm prog.o  
$ cc prog.o -o prog
```

Parametrus, ar kuriem, **gcc** izsauc **ld** var uzzināt:

```
$ cc -v get_volume_libc.o -o get_volume_libc  
Using built-in specs.  
Target: i386-undermydesk-freebsd  
Configured with: FreeBSD/i386 system compiler  
Thread model: posix  
gcc version 4.2.1 20070719 [FreeBSD]  
/usr/bin/ld --eh-frame-hdr -V -dynamic-linker /libexec/ld-elf.so.1 -o  
get_volume_libc /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/crtbegin.o -L/usr/lib -L/usr/lib  
get_volume_libc.o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed  
-lgcc_s --no-as-needed /usr/lib/crtend.o /usr/lib/crtn.o  
GNU ld version 2.15 [FreeBSD] 2004-05-23  
Supported emulations:  
elf_i386_fbsd
```

# Hello world ar libc

format ELF

extrn write

extrn exit

STDOUT = 1

EXIT\_SUCCESS = 0

section '.text' executable

public main

main:

push msg\_len

push msg

push STDOUT

call write

add esp,3\*4 ; clear stack

push EXIT\_SUCCESS

call exit

section '.data' writeable

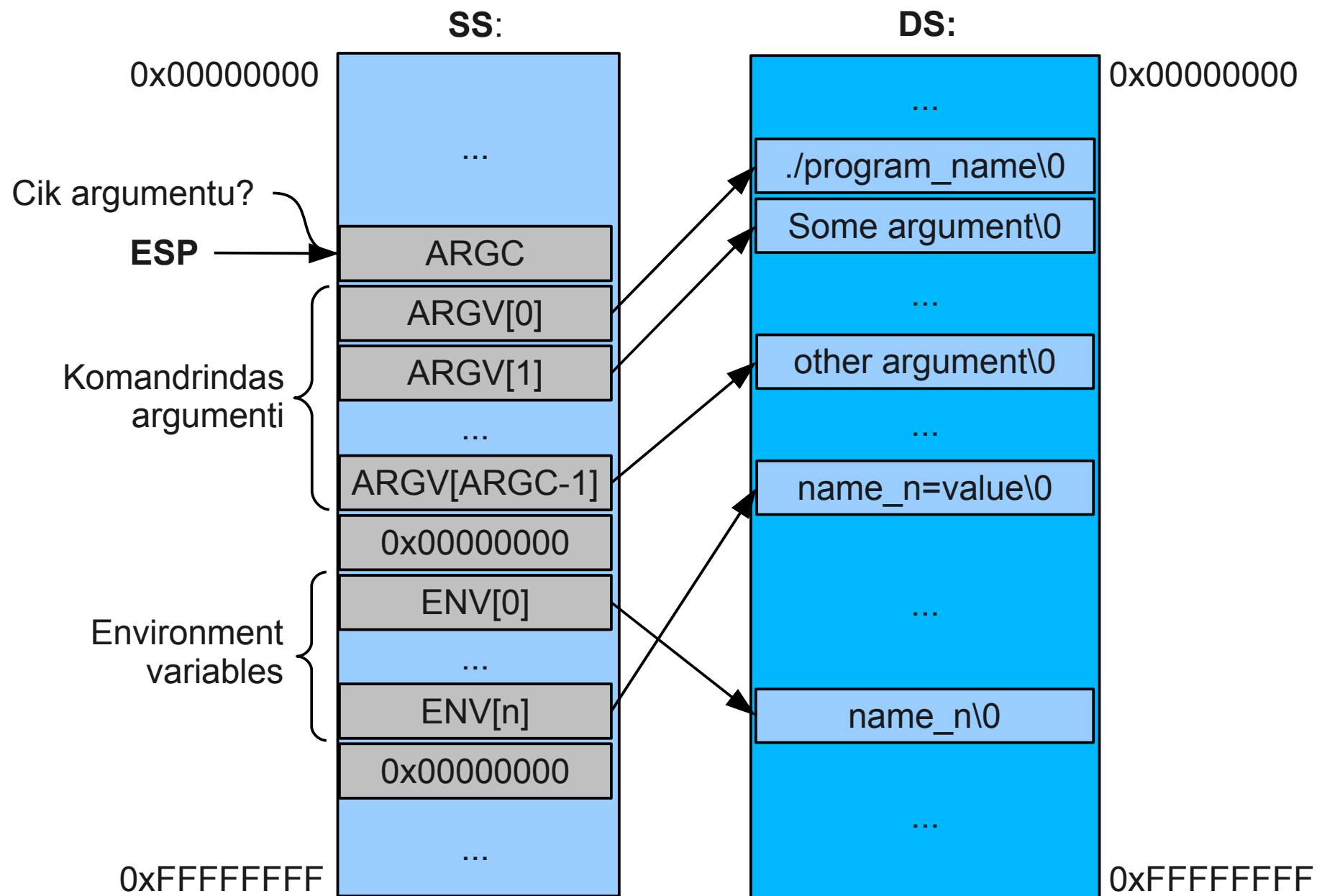
msg db "Hello world",0xA

msg\_len = \$-msg

*Šajā piemērā kļūdas netiek pārbaudītas*

Šo programmu varēs nokompilēt un palaist uz jebkuras 32 bitu operētājsistēmas, kurai ir libc bibliotēka. Pie tam šo programmu var palaist arī uz 64 bitu operētājsistēmām, kuras atbalsta 32 bitu programmas (un arī ir 32 bitu libc).

# Kas ir ielikts stekā?



*Vides mainīgiem vērtība var arī būt nedefinēta (šajā gadījumā "=" var nebūt)*



# Krāsaina teksta izvade :)

\e[...;...;...m

Foreground colors

30 Black

31 Red

32 Green

33 Yellow

34 Blue

35 Magenta

36 Cyan

37 White

39 Default

Background colors

40 Black

41 Red

42 Green

43 Yellow

44 Blue

45 Magenta

46 Cyan

47 White

49 Default

\ec – RESET terminal

\e[K – Erase line

\e[2J – Erase screen

\e[line;columnH – move cursor to pos

Piemēram:

\e[2J\e[33;44mHello World\n\e[39;49m

Notīrīs ekrānu un

izvadīs Hello World dzeltenā krāsā uz zila fona

Text attributes

0 All attributes off

1 Bold on

4 Underscore (on monochrome display adapter only)

5 Blink on

7 Reverse video on

8 Concealed on

<http://ascii-table.com/ansi-escape-sequences.php>

<http://ascii-table.com/ansi-escape-sequences-vt-100.php>

# Darbs ar iekārtām

Unix vidē faili ir faili, direktorijas ir faili un iekārtas (ar retiem izņēmumiem) ir faili (speciāli faili) :)

Visas iekārtas (kas ir faili) atrodas **/dev/** direktorijā  
Ir divu veidu iekārtas simbolu (character devices) un bloku (block devices) iekārtas.

Ar iekārtām darbs notiek tās atverot kā failus ar `SYS_OPEN` kodola izsaukumu.

Tālākais darbs lielākoties notiek ar `SYS_IOCTL` kodola izsaukumu vai ar `SYS_READ/SYS_WRITE`.

# Vienkāršs piemērs: skaņas iekārtu izmantošana

Visvienkāršākā iekārta ir **/dev/sndstat**. Šo iekārtu var tikai nolasīt... tā satur informāciju par skaņas kartes draiveru. Šī iekārta gan programmētājam nav noderīga :)

**/dev/dsp** (digital signal processor) ir svarīgākā skaņas iekārta. Tajā var rakstīt un to var nolasīt. Ja **/dev/dsp** atvērt lasīšanai, tad var nolasīt datus no mikrofona... :D Ja atvērt **/dev/dsp** rakstīšanai, tad var izvadīt datus pa skaļruņiem. Šo iekārtu jāatver vainu lasīšanai, vai rakstīšanai, bet ne rakstīšanai un lasīšanai.

**/dev/mixer** kontrolē dažādus skaļuma parametrus. Darbs ar šo iekārtu notiek ar **SYS\_IOCTL** kodola izsaukumu. Ja ir atvērts **/dev/dsp**, **/dev/mixer** var arī neizmantot. Šajā gadījumā viss, kas attiecas uz **/dev/mixer** ir spēkā arī **/dev/dsp**

Konstantes darbam ar **SYS\_IOCTL** var atrast failā:

- **/usr/include/sys/soundcard.h** (uz BSD)
- **/usr/include/linux/soundcard.h** (uz Linux)

# Sīkāka informācija

<http://int80h.org/> - Ka programmēt assembler uz FreeBSD

<http://www.exforsys.com/tech-articles/unix/linux-assembly-tutorial-step-by-step-guide.html>

- Ka programmēt assembler uz Linux

**Зубков С.В. - “Assembler для DOS, Windows и UNIX”, Москва, 2005 ISBN 5-94075-259-9**

– Vispārēja informācija par programmēšanu assembler. Ļoti laba rokasgrāmata. Izcils satura rādītājs. Par assembler programmēšanu UNIX un Windows vidē gan, tur praktiski nekā nav.

<http://redir.no-ip.org/mirrors/my.execpc.com/~geezer/osd/index.htm> - interesants resurss.

Apraksta dažādus OS programmēšanas aspektus

Jautājumi?