6η Εργαστηριακή Άσκηση για το μάθημα Σχεδιασμός Ενσωματωμένων Συστημάτων

Ομάδα 29

Μέλη : Αριστοτέλης Γρίβας

Σαββας Λεβεντικίδης

Προετοιμασία για την άσκηση

Για την ορθή πραγματοποίηση της άσκησης θα χρειαστεί να ακολουθήσουμε τα παρακάτω βήματα:

1.

Αρχικά, θα πρέπει να δημιουργήσουμε ένα δεύτερο εικονικό μηχάνημα (Virtual Machine) στο QEMU. Προκειμένου να συμβεί αυτό, θα χρειαστεί να κατεβάσουμε τα παρακάτω αρχεία (αντίστοιχα με τα αρχεία της εργαστηριακής άσκησης 5):

- https://people.debian.org/~aurel32/qemu/armhf/debian_wheezy_armhf_standard.qcow2
- https://people.debian.org/~aurel32/qemu/armhf/initrd.img-3.2.0-4-vexpress
- https://people.debian.org/~aurel32/qemu/armhf/vmlinuz-3.2.0-4-vexpress

2.

Αντίστοιχα με την 5η εργαστηριακή άσκηση μπορούμε να ξεκινήσουμε το εικονικό μας μηχάνημα εκτελώντας την παρακάτω εντολή:

```
sudo qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd initrd.img-3.2.0-4-vexpress-driveif=sd,file=debian_wheezy_armhf_standard.qcow2-append"root=/dev/mmcblk0p2" -net nic -net user,hostfwd=tcp:127.0.0.1:22223-:22
```

Στο σημείο αυτό μας εμφανίζεται το error :

*qemu-system-arm: Invalid SD card size: 25 GiB SD card size has to be a power of 2, e.g. 32 GiB

Που σημαίνει πως πρέπει να κάνουμε resize το image που κατεβάσαμε, προκειμένου το μέγεθος να γίνει ίσο με κάποια δύναμη του 2 μεγαλύτερη από το μέγεθος του αρχείου που. Συνεπώς, εκτελούμε την εντολή :

```
qemu-img resize debian_wheezy_armhf_standard.qcow2 32G
```

3.

Επίσης, θα χρειαστεί να ανανεώσουμε το αρχείο sources.list, προκειμένου να έχουμε πρόσβαση στα πακέτα των repositories του Debian και να ανανεώσουμε τις λίστες πακέτων (apt-get update), όπως κάναμε και για το προηγούμενο εικονικό μας μηχάνημα.

Εγκατάσταση custom cross-compiler building toolchain crosstool-ng

Τα βήματα που ακολουθήσαμε για την εγκατάσταση του crosstool-ng είναι όπως αναφέρονται στην εκφώνηση, και ακολουθούν ως εξής :

1.

Αρχικά θα πρέπει να κατεβάσουμε τα απαραίτητα αρχεία για το κτίσιμο του toolchain από το αντίστοιχο github repository στο host μηχάνημά. Εκτελούμε την εντολή

```
:~$ git clone https://github.com/crosstool-ng/crosstool-ng.git
```

Η εντολή δημιουργεί στο directory που την εκτελέσαμε έναν φάκελο με όνομα crosstool-ng.

2.

Μπαίνουμε στο φάκελο, και αρχικά εκτελούμε

```
:~/crosstool-ng$ ./boostrap
```

3.

Στη συνέχεια, δημιουργούμε δύο φακέλους στο HOME directory μας ως εξής :

```
:~/crosstool-ng$ mkdir $HOME/crosstool && mkdir $HOME/src
```

Στον πρώτο φάκελο θα εγκατασταθεί το πρόγραμμα crosstool-ng ενώ στον δεύτερο, θα αποθηκεύει τα απαραίτητα πακέτα που κατέβαζει για να χτίσει τον cross-compiler.

4.

Εκτελούμε την παράκατω εντολή για να κάνουμε configure την εγκατάσταση του crosstool-ng:

```
:~/crosstool-ng$ ./configure --prefix=${HOME}/crosstool
```

Κατά τη διάρκεια της εκτέλεσης αυτής της εντολής εμφανίστηκαν διάφορα πακέτα που έλλειπαν. Συνεπώς χρειάστηκε να εγκαταστήσουμε τα ακόλουθα :

Sudo apt-get install :

- build-essential
- gawk
- flex
- texinfo
- libtool libtool-bin libtool-doc
- bison
- ncurses-dev
- help2man
- byacc
- automake
- autoconf
- gettext
- autopoint (είναι μέρος του gettext)

Αφού κατεβάσουμε τα παραπάνω packages, επαναλαμβάνουμε την εντολή για το configure, και εκτελείται χωρίς κανένα σφάλμα.

5.

Εκτελούμε την εντολή make και make install:

```
:~/crosstool-ng$ make && make install
```

6.

Στο σημείο αυτό, έχει εγκατασταθεί το crosstool-ng. Πηγαίνουμε στο installation path $HOME/crosstool/bin.

```
:~/crosstool-ng$ cd $HOME/crosstool/bin
```

Όπου και θα κάνουμε build τον custom compiler μας.

7.

Εκτελούμε την εντολή

```
:~/crosstool/bin$ ./ct-ng list-samples
```

Εμφανίζεται μία λίστα με πολλούς συνδυασμούς αρχιτεκτονικών, λειτουργικών συστημάτων και βιβλιοθηκών της C που παρέχονται από το εργαλείο για να μπορούμε να κάνουμε γρήγορα και σωστά configure το build του cross-compiler που

θέλουμε να παράξουμε για συγκεκριμένο target machine. Εμείς επιλέγουμε την: ***arm-cortexa9_neon-linux-gnueabihf***.

8.

Εκτελούμε την εντολή

:~/crosstool/bin$ ./ct-ng arm-cortexa9_neon-linux-gnueabihf

για να παραμετροποιήσουμε το crosstool-ng για τη συγκεκριμένη αρχιτεκτονική

9.

Αν θέλουμε να κάνουμε κάποια αλλαγή στο configuration του target machine, ακολουθούμε την εξής εντολή :

:~/crosstool/bin$ ./ct-ng menuconfig

Ωστόσο, στα πλαίσια της άσκησης αφήσαμε το default configuration.

10.

Τέλος αφού έχουμε παραμετροποιήσει τον cross compiler, κάνουμε build. Εκτελούμε:

:~/crosstool/bin$ ./ct-ng build

Και δημιουργείται ο φάκελος $HOME/x-tools/arm-cortexa9_neonlinux-gnueabihf όπου μέσα στον υποφάκελο bin περιέχει τα εκτελέσιμα αρχεία του cross compiler μας.

Εγκατάσταση pre-compiled building toolchain linaro

Εκτός από τη χρήση του custom compiler toolchain, θα χρησιμοποιήσουμε και έναν pre-compiled cross compiler που παρέχεται από την ιστοσελίδα www.linaro.org/downloads . Για να κάνουμε χρήση του pre-compiled cross compiler εκτελούμε τα παρακάτω βήματα:

1.

Κατεβάζουμε τα binaries του cross compiler στο host μηχάνημα από την παρακάτω διεύθυνση:

:~$ mkdir ~/linaro && cd ~/linaro

:~/linaro$

wget https://releases.linaro.org/archive/14.04/components/toolchain/binaries/gcclinaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz2

2.

Κάνουμε extract τα αρχεία που κατεβάσαμε:

```
:~/linaro$ tar -xvf gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz2
```

3.

Τα binaries του cross compiler βρίσκονται στο πακέτο που κατεβάσαμε στον φάκελο $HOME/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin

Άσκηση 1:

1.

Στα πλαίσια της άσκησης κατεβάσαμε το image debian_wheezy_armhf, ενώ στην άσκηση 5 είχαμε χρησιμοποιήσει το image debian_wheezy_armel. Τα δύο αυτά images παρουσιάζουν τις εξής διαφορές :

ARMHF (ARM Hard Float):

Αυτή η έκδοση του Debian προορίζεται για αρχιτεκτονικές arm από ARMv7 και πάνω, οι οποίες χρησιμοποιούν hardware floating-point units (FPU). Αυτό σημαίνει πως στις αρχιτεκτονικές αυτές οι υπολογισμοί floating-point γίνονται εξ'ολοκλήρου στο hardware, οδηγώντας σε ταχύτερους υπολογισμούς floating-point λειτουργιών. Συνεπώς, ένα τέτοιο image απευθύνεται κυρίως σε νεότερες αρχιτεκτονικές arm με τις παραπάνω δυνατότητες.

ARMEL (ARM Little-Endian):

Αυτή η έκδοση του Debian προορίζεται για παλαιότερες αρχιτεκτονικές (π.χ. ARMv5, Armv6), οι οποίες δεν έχουν ενσωματωμένα FPUs, με αποτέλεσμα όλες οι λειτουργίες floating-point να μη γίνονται στο hardware, αλλά να τις διαχειρίζεται το software, κάτι το οποίο μπορεί να οδηγήσει σε μειωμένη απόδοση. Ωστόσο, οι armel διανομές, είναι συμβατές με ένα ευρύτερο φάσμα arm επεξεργαστών, και η ιδανική επιλογή για αυτούς οι οποίοι δεν έχουν υποστήριξη για floating-point λειτουργίες.

Συμπερασματικά :

Χρησιμοποιούμε armhf για νεότερες αρχιτεκτονικές οι οποίες έχουν λειτουργίες floating-point και απαιτείται υψηλή επίδοση, ενώ armel για παλαιότερες αρχιτεκτονικές, όπου έχουμε ανάγκη το compatibility της έκδοσης, και όπου δεν υποστηρίζονται floating-point λειτουργίες. Τονίζουμε πως οι αρχιτεκτονικές Armv7 και άνω (όπου χρειαζόμαστε armhf), υποστηρίζουν μέχρι 64-bit instruction sets, ενώ οι παλαιότερες αρχιτεκτονικές (όπου χρειαζόμαστε armel) υποστηρίζουν μέχρι 32-bit instruction sets.

2.

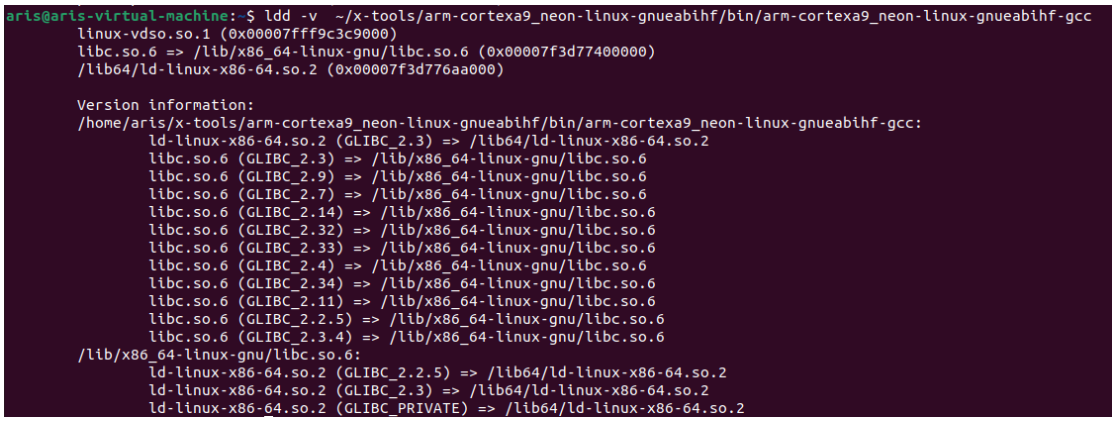
Χρησιμοποιούμε την αρχιτεκτονική arm-cortexa9_neon-linux-gnueabihf, ώστε να υπάρχει συμβατότητα μεταξύ της αρχιτεκτονικής και του συστήματος που έχουμε στο qemu. Αναλυτικότερα, όπως φανερώνει και το όνομα, η αρχιτεκτονική προορίζεται για λειτουργικό βασισμένο σε Linux στο target μηχάνημα, καθώς και για το interface GNU EABI (Embedded Application Binary Interface), το οποίο υποστηρίζει υπολογισμούς floating-point (gnueabi**hf**). Ο cortex-a9 χρησιμοποιεί armv7-a επεξεργαστή, που όπως περιγράψαμε παραπάνω, κάνει όλο και πιο φανερή την ανάγκη για υποστήριξη floating-point. Καταλαβαίνουμε, λοιπόν, πως η αρχιτεκτονική αυτή αποτελεί μία ιδανική επιλογή λόγω του compatibility που προσφέρει με το εν λόγω σύστημα, το οποίο τρέχει σε Linux, καθώς και της ανάγκης για floating-point operations .

3.

Στο 9ο βήμα μας δόθηκε η δυνατότητα να αλλάξουμε το configuration, και να χρησιμοποιήσουμε κάποια άλλη βιβλιοθήκη, όμως εμείς συνεχίσαμε με τη default επιλογή της glibc. Αυτή η επιλογή βασίστηκε στα εξής :

- Η glibc είναι η πλέον χρησιμοποιούμενη βιβλιοθήκη της C σε Linux συστήματα, και μας εξασφαλίζει συμβατότητα με ένα ευρύτερο φάσμα από εφαρμογές σε Linux.
- Παρόλο που δεν είναι η γρηγορότερη βιβλιοθήκη της C, παρέχει αρκετά καλή επίδοση για υπολογιστικές λειτουργίες γενικού σκοπού.
- Η συγκεκριμένη βιβλιοθήκη αποτελεί τη standard επιλογή, κάτι το οποίο σημαίνει πως έχουμε δυνατότητα φοριτότητας και διαλειτουργηκότητας με άλλες πλατφόρμες βασισμένες σε Linux, κάτι το ιδανικό για cross-compilation.

Επομένως, παρόλο που δεν είναι η γρηγορότερη ή η βέλτιστη για κάποια μεμονωμένη εφαρμογή, η glibc αποτελεί την ασφαλέστερη επιλογή για το σύστημά μας. Με χρήση της εντολής ldd μπορούμε να δούμε τη βιβλιοθήκη που χρησιμοποιούμε :



4.

Σε αυτό το βήμα, κάνουμε compile των κώδικα phods που δόθηκε σε προηγούμενη άσκηση, χρησιμοποιώντας τον cross-compiler του crosstool-ng.

$/x-tools/arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon- linux-gnueabihf-gcc -00 -Wall -o phods_crosstool.out phods.c

Εκτελούμε, τώρα, τις εξής εντολές :

file:

```
aris@aris-virtual-machine:~/Desktop$ file phods_crosstool.out
phods_crosstool.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0
, with debug_info, not stripped
```

Αυτή η εντολή μας δίνει πληροφορίες για τα χαρακτηριστικά του εκτελέσιμου αρχείου που έχουμε. Αναλυτικότερα, βλέπουμε ότι το αρχείο μας είναι αρχείο ELF(Executable and Linkable Fromat), εκτελέσιμο για 32-bit little-endian σύστημα, προορίζεται για αρχιτεκτονική arm, με 5^{η} έκδοση του EABI (EABI5). Επιπλέον, είναι δυναμικά Linked, βασίζεται επομένως σε shared libraries κατά τη διάρκεια της εκτέλεσης, όπου ο dynamic linker δίνεται στη συνέχεια (ld-linux-armhf.so.3). Τέλος, έχουμε την πληροφορία πως το εκτελέσιμο προορίζεται για GNU/Linux σύστημα έκδοσης 3.2.0 και νεότερης, περιέχει πληροφορίες σχετικά με το debugging, και δεν είναι stripped, δηλαδή δεν έχουν αποβληθεί πληροφορίες από το αρχείο για να μειωθεί το μεγεθός του.

readelf:

```
aris@aris-virtual-machine:~/Desktop$ readelf -h -A phods_crosstool.out
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x10478
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14972 (bytes into file)
  Flags:                             0x5000400, Version5 EABI, hard-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         36
  Section header string table index: 35
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "7-A"
  Tag_CPU_arch: v7
  Tag_CPU_arch_profile: Application
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-2
  Tag_FP_arch: VFPv3
  Tag_Advanced_SIMD_arch: NEONv1
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_rounding: Needed
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align_needed: 8-byte
  Tag_ABI_align_preserved: 8-byte, except leaf SP
  Tag_ABI_enum_size: int
  Tag_ABI_VFP_args: VFP registers
  Tag_CPU_unaligned_access: v6
  Tag_MPextension_use: Allowed
  Tag_Virtualization_use: TrustZone
```

Αυτή η εντολή μας δίνει πληροφορίες σχετικά με τα Elf headers και sections του εκτελέσιμου αρχείου. Παραθέτει λεπτομέρειες σχετικά με τον τύπο αρχείου, την αρχιτεκτονική , τις δυνατότητες floating point και το ABI. Το συγκεκριμένο εργαλείο βοηθά , γενικότερα, στην εποπτεία των binary files και του debugging.

5.

Στη συνέχεια, κάνουμε και πάλι compile τον κώδικα, αυτή τη φορά ωστόσο, με τον cross-compiler που κατεβάσαμε από το site της linaro, με την παρακάτω εντολή:

$~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin/arm-linux-gnueabihf-gcc -00 -Wall -0 phods_linaro.out phods.c

Επίσης χρειαστήκαμε να κατεβάσουμε το εξής package : ***lib32z1-dev***.

Παρατηρούμε πως τα μεγέθη των αρχείων είναι τα εξής :

```
-rwxrwxr-x  1 aris aris  17K Φεβ  17 21:14 phods_crosstool.out
-rwxr-xr-x  1 root root 8,1K Φεβ  17 21:44 phods_linaro.out
```

Παρατηρούμε πως το εκτελέσιμο που παράγεται από τον cross-compiler του linaro είναι αρκετά μικρότερο από αυτό του custom cross compiler, και αυτό είναι λογικό καθώς η έκδοση της glibc που χρησιμοποιεί το Linaro είναι αυτή των 32-bit , σε αντίθεση με αυτή των 64-bit που χρησιμοποιεί ο custom cross-compiler.

6.

Το πρόγραμμα εκτελείται σωστά στο target μηχάνημα, αφού 32-bit εκτελέσιμα μπορούν να εκτελεστούν σωστά σε 64-bit μηχανήματα.

7.

Τώρα κάνουμε και πάλι compile τον κώδικα phods με τον custom cross-compiler, καθώς και τον cross-compiler της linaro, αλλά αυτή τη φορά προσθέτουμε το flag -static. Το flag που προσθέσαμε ζητάει από τον εκάστοτε compiler να κάνει στατικό linking της αντίστοιχης βιβλιοθήκης της C του κάθε compiler. Παρατηρούμε και πάλι τα μεγέθη των εκτελέσημων:

```
-rwxrwxr-x  1 aris aris 2,8M Φεβ  17 21:50 phods_crosstool.out
-rwxr-xr-x  1 root root 497K Φεβ  17 21:50 phods_linaro.out
```

Βλέπουμε πως τα εκτελέσιμα αρχεία έχουν πολύ μεγαλύτερο μέγεθος αυτή τη φορά. Αυτό οφείλεται στο γεγονός πως το flag -static, περιλαμβάνει όλο τον κώδικα της βιβλιοθήκης απευθείας στα εκτελέσιμα αρχεία. Είναι σημαντικό να τονίσουμε πως παρόλο που το μέγεθος αυξάνεται σημαντικά, το στατικό linking μπορεί να οδηγήσει σε αποφυγή προβλημάτων συμβατότητας καθώς και σε πλεονεκτήματα που έχουν να κάνουν με τη φορητότητα του εκτελέσιμου. Σε κάθε περίπτωση, η επιλογή της χρήσης ή μη του flag εξαρτάται από τους συμβιβασμούς που μπορεί να κάνει ο χρήστης. Τέλος, παρατηρούμε πως το εκτελέσιμο του linaro cross-compiler παραμένει μικρότερο σε σύκγριση με αυτό του custom cross-compiler.

8.

Ακολουθούν τα εξής υποθετικά ερωτήματα :

Προσθέτουμε μία δική μας συνάρτηση στη mlab_foo() στη glibc και δημιουργούμε έναν crosscompiler με τον crosstool-ng που κάνει χρήση της ανανεωμένης glibc. Δημιουργούμε ένα αρχείο my_foo.c στο οποίο κάνουμε χρήση της νέας συνάρτησης που δημιουργήσαμε και το κάνουμε cross compile με flags -Wall -O0 -o my_foo.out

A. Τι θα συμβεί αν εκτελέσουμε το my_foo.out στο host μηχάνημα;

B. Τι θα συμβεί αν εκτελέσουμε το my_foo.out στο target μηχάνημα;

C. Προσθέτουμε το flag -static και κάνουμε compile ξανά το αρχείο my_foo.c. Τι θα συμβεί τώρα αν εκτελέσουμε το my_foo.out στο target μηχάνημα;

Απάντηση :

A. Αν εκτελέσουμε το my_foo.out στο host μηχάνημα, αυτό **δεν θα εκτελεστεί** καθώς έχει φτιαχτεί για την cross-compiled αρχιτεκτονική του target μηχανήματος και όχι αυτή του host.
B. Αν εκτελέσουμε το my_foo.out στο target μηχάνημα, αυτό **δεν θα εκτελεστεί** και πάλι, γιατί χρησιμοποιεί την default έκδοση της βιβλιοθήκης glibc, η οποία δεν περιέχει τη συνάρτηση mlab_foo().
C. Αν προσθέσουμε το flag -static και κάνουμε compile ξανά το αρχείο my_foo, τότε αν εκτελέσουμε το my_foo.out στο target μηχάνημα, αυτό **θα εκτελεστεί κανονικά**, γιατί το flag -static θα έχει συμπεριλάβει όλο τον κώδικα της βιβλιοθήκης glibc, μαζί με τη νέα συνάρτηση, όπως ακριβώς αναφέραμε και στο προηγούμενο ερώτημα.

Άσκηση 2:

1.

Για την συγκεκριμένη άσκηση θα χρησιμοποιήσουμε τον cross compiler που εγκαταστήσαμε στην αρχή της παρούσας εργαστηριακής άσκησης και συγκεκριμένα το crosstool-ng.

2.

Με σκοπό να μπορέσουμε στο τέλος της άσκησης να ξεχωρίσουμε και να συγκρίνουμε τα αποτελέσματα των δύο διαφορετικών kernel αποθηκεύουμε στο host μηχάνημα το boot directory του current qemu kernel με την εξής εντολή:

```
$ scp -P 22223 -r root@localhost:/boot ./
```

3.

Κατεβάζουμε τον source code του new kernel με τις παρακάτω εντολές

```
root@qemu:~$ apt-get update
```

```
root@qemu:~$ apt-get install linux-source
```

Η εκτέλεση αυτή θα κατεβάσει στο directory /usr/src το αρχείο linux-source-3. 16.tar. xz

4.

Το compiling του πυρήνα θα γίνει στο host μηχάνημα γιατί ο χρόνος που χρεάζεται για να γίνει στο guest μηχάνημα είναι απαγορευτικός. Αντιγράφουμε λοιπόν στο host μηχάνημα το αρχείο που κατεβάσαμε και το κάνουμε extract με τις παρακάτω εντολές

```
$ scp -P 22223 -r root@localhost:/usr/src/linux-source-3.16.tar.xz ./
```

```
$ tar -xf linux-source-3.16.tar.xz
```

5.

Στα πλαίσια της άσκησης μας δίνεται ένα προκαθορισμένο kernel configuration, το οποίο μπορούμε να βρούμε στο helios με το όνομα kernel_config. Για να χρησιμοποιήσουμε το έτοιμο configuration, θα πρέπει να το κατεβάσουμε στον υπολογιστή μας και να το τοποθετήσουμε στο directory του linux-source ως .config με την εξής εντολή :

```
:~/path/to/kernel$ cp /path/to/kernel_config .config
```

,όπου path to kernel ο untarred φάκελος του new kernel source code και path to kernel config η θέση του αρχείου που κατεβάσαμε από το helios.

Επιπλέον, κατεβάζουμε επιπλέον το builddeb_hf.patch που δίνεται στο helios και τροποποιούμε αρχείο στο scripts/package/builddeb, ώστε να είναι patched.

Επειδή κάνουμε cross-compiling, υποδείξαμε στο make την target αρχιτεκτονική και τον cross compiler που θα χρησιμοποιήσουμε. Έτσι θα χτίσουμε τον πυρήνα του linux εκτελώντας την παρακάτω εντολή:

```
:~/path/to/kernel$ make ARCH=arm CROSS_COMPILE=<path_to_your_cross_compiler>/bin/arm-cortexa9_neon-linux-gnueabihf
```

,όπου path to your cross_compiler είναι το /home/username/x-tools/arm-cortexa9_neon-linux-gnueabihf (αν έχει ακολουθηθεί σωστά η διαδικασία στην αρχή της εργαστηριακής άσκησης)

Κατά την εκτέλεση της εντολής αυτής εμφανίστηκε το εξής σφάλμα:

“multiple definition of `yylloc';

το οποίο προσπεράσαμε τροποποιώντας το αρχείο :

/path/to/linuxsource/scripts/dtc/dtc.lexer.lex.c_shipped

δηλώνοντας την μεταβλητή YYLTYPE yylloc ως extern.

Εκτελούμε και πάλι την εντολή make.

Κατά την εκτέλεση αυτής της εντολής παρουσιάζεται εκ νέου το εξής σφάλμα :

arch/arm/mm/proc-v7.S: Assembler messages:

arch/arm/mm/proc-v7.S:429: Error: junk at end of line, first unrecognized character is `#'

και η γραμμή στην οποία παρουσιάζεται το σφάλμα είναι η εξής :

.section ".proc.info.init", #alloc, #execinstr

Παρατηρούμε πως ο assembler φαίνεται να έχει κάποιο πρόβλημα με το χαρακτήρα #, οπότε σε πρώτη φάση τον αφαιρούμε και παρατηρούμε τη συμπεριφορά. Έχουμε και πάλι το σφάλμα :

arch/arm/mm/proc-v7.S: Assembler messages:

arch/arm/mm/proc-v7.S:429: Error: junk at end of line, first unrecognized character is `a'

όπου βλέπουμε πως ο assembler συνεχίζει να έχει το ίδιο πρόβλημα. Συνεπώς, κατανοούμε πως πρέπει να βρούμε ένα τρόπο για να ορίσουμε τα options #alloc (δηλ. allocatable section) και #execinstr (δηλ. executable section). Αντικαθιστούμε, λοιπόν, τα options αυτά με τα αντίστοιχα flags, “ax” (allocatable,executable). Τελικά, η γραμμή κώδικα είναι η εξής :

.section ".proc.info.init", “ax”

Εκτελούμε ξανά την εντολή make, και παρατηρούμε πως δεν αντιμετωπίσαμε πρόβλημα στο αρχείο arch/arm/mm/proc-v7.S, όπου σταατούσε το make προηγουμένως. Ωστόσο, εμφανίστηκε το ίδιο σφάλμα για άλλο αρχείο. Έτσι, καταλαβαίνουμε πως η αλλαγή αυτή πρέπει να πραγματοποιηθεί για όλα τα αρχεία τα οποία περιέχουν τις εξής γραμμές :

- #alloc, #execinstr
- #alloc
- #execinstr

Έτσι χρησιμοποιούμε τις παρακάτω εντολές αντίστοιχα:

- find ~/linux-source3.16/ -type f -exec sed -i 's/ #alloc, #execinstr/ “ax”/g' {} +
- find ~/linux-source3.16/ -type f -exec sed -i 's/ #alloc/ “a”/g' {} +
- find ~/linux-source3.16/ -type f -exec sed -i 's/ #execinstr/ “x”/g' {} +

Οι εντολές αυτές χρησιμοποιούν την εντολή find για να βρουν όλα τα αρχεία στο directory **~/linux-source3.16**, τα οποία περιέχουν τα :

“#alloc” και “#execinstr”

και τα αντικαθιστούν με τα :

“a” και “x” αντίστοιχα, ή “ax” αν υπάρχουν και τα δύο μαζί στην ίδια γραμμή.

Στη συνέχεια, τρέχουμε πάλι την εντολή make, και παρατηρούμε πως δεν παρουσιάζεται κανένα σφάλμα κατά την εκτέλεση.

6.

Τώρα θα δώσουμε οδηγία στο make να δημιουργήσει 3 *.deb πακέτα με το image του πυρήνα, τα ανανεωμένα kernel headers και ένα ακόμα πακέτο με headers από τον πυρήνα του Linux που χρησιμοποιούνται για userspace προγραμματισμό, μέσω την libc και των βιβλιοθηκών συστήματος. Για να συμβουν τα παραπάνω πρέπει να δώσουμε στο make το directive deb-pkg. Τελος με το flag -j και έναν αριθμό μεγαλύτερο του 1 μπορούμε να εκτελέσουμε το make σε παραπάνω από έναν επεξεργαστές για να επιταχυνθεί η διαδικάσια. Συνολικά η ελάχιστη μορφή της εντολής που χρησιμοποιήσαμε είναι :

```
:~/path/to/kernel$ make ARCH=arm
CROSS_COMPILE=<path_to_your_cross_compiler>/bin/arm-
cortexa9_neon-linux-gnueabihf deb-pkg -j 2
```

Τα αρχεία .deb παράγονται με επιτυχία.

Στο σημείο ανεβάζουμε τα αρχεία .deb στο guest μηχάνημα με την εντολή scp όπως πριν, και επιχειρούμε να τα εγκαταστήσουμε με χρήση του packet manager μέσω της εντολής (έχοντας ενημερώσει τα πακέτα dpkg και bzip2):

root@qemu:~$ dpkg -i package_name.deb

ωστόσο λαμβάνουμε το εξής σφάλμα :

dpkg-deb: error: archive 'linux-headers-3.16.84_3.16.84-4_armhf.deb' uses unknown compression for member 'control.tar.zst', giving up

Το σφάλμα αυτό μας φανερώνει πως η έκδοση του dpkg-deb που έχει το σύστημά μας δεν μπορεί να υποστηρίξει decompression σε αρχεία .zst. Σε πρώτη φάση, λοιπόν, πρέπει να εγκαταστήσουμε το zstd πακέτο που χρησιμοποιείται για τέτοιο είδους decompression.

Και πάλι, ωστόσο, αντιμετωπίζουμε άλλο ένα πρόβλημα καθώς (πιθανότατα λόγω του αρχείου sources.list και των πακέτων τα οποία υποστηρίζει για installation η συγκεκριμένη έκδοση του Debian), δεν μπορούμε να εγκαταστήσουμε το πακέτο μέσω :

```
~:apt-get install zstd
```

Επομένως, καταφεύγουμε στο να κάνουμε Install το πακέτο αυτό manually. Έτσι κάνουμε τα εξής βήματα :

1. Μεταφερόμαστε στο **host** μηχάνημα για να έχουμε πιο γρήγορα εκτέλεση και με λιγότερα σφάλματα.
2. Κατεβάζουμε το source code του zstd πακέτου μέσω της εντολής :
   wget https://github.com/facebook/zstd/archive/refs/tags/v1.5.0.tar.gz
3. Κάνουμε decompress το αρχείο tar μέσω της εντολής :
   tar -xzvf v1.5.0.tar.gz
   και έτσι δημιουργείτε το directory με όνομα “zstd-1.5.0”.
4. Στη συνέχεια μεταφέρουμε τα περιεχόμενα του directory αυτού στο guest μηχάνημα μέσω της εντολής scp.
5. Και τώρα είμαστε έτοιμοι για το installation του zstd στο guest μηχάνημα. Μπαίνουμε στο directory όπου αποθηκεύσαμε το zstd-1.5.0 στο guest μηχάνημα και εκτελούμε **make && make-install**.
6. Το zstd έχει εγκατασταθεί με επιτυχία στο guest μηχάνημα.

Δοκιμάζουμε και πάλι την εντολή :

root@qemu:~$ dpkg -i package_name.deb

όμως και πάλι έχουμε το ίδιο σφάλμα με πριν καθώς, όπως αναφέραμε, το dpkg δεν υποστηρίζει decompression .zst αρχείων, ακόμα και ας έχουμε εγκατεστημένο το zstd.

Άρα και πάλι παρατηρούμε πως το decompression πρέπει να πραγματοποιηθεί manually. Για το λόγο αυτό, δημιουργούμε ένα δικό μας bash script, με όνομα comp.sh. Ο κώδικας είναι ο εξής :

```
#!/bin/bash

DEBPACKAGE="${1%.deb}"

[[ -z "$1" ]] && echo "Usage: $0 <package.deb>" && exit 1

set -e
ar x $DEBPACKAGE.deb
zstd -d < control.tar.zst | xz > control.tar.xz
zstd -d < data.tar.zst | xz > data.tar.xz
ar -m -c -a sdsd "$DEBPACKAGE"_repacked.deb debian-binary control.tar.xz data.tar.xz
rm debian-binary control.tar.xz data.tar.xz control.tar.zst data.tar.zst
echo "Repack done. Use the following command to install."
echo "sudo dpkg -i --force-overwrite ${DEBPACKAGE}_repacked.deb
```

Ο κώδικας αυτός παίρνει ως όρισμα ένα αρχείο .deb και κάνει τα εξής :

1. Αφαιρεί το extension .deb για να πάρει το όνομα του βασικού πακέτου.
2. Αποκτά τα περιεχόμενα του .deb πακέτου μέσω της εντολής " ar".
3. Κάνει decompress τα αρχεία 'control.tar.zst' και 'data.tar.zst' (μέσω του zstd που εγκαταστήσαμε) και στη συνέχεια τα κάνει ξανά compress σε μορφή 'control.tar.xz' και 'data.tar.xz', τα οποία μπορεί να τα χειριστεί το dpkg.
4. Στη συνέχεια δημιουργεί ένα νέο .deb πακέτο, το οποίο πλέον μπορούμε να το χειριστούμε με την εντολή dpkg.

Εκτελούμε λοιπόν για τα 3 .deb αρχεία την εντολή :
./comp.sh file_name.deb

Τώρα μπορούμε να εκτελέσουμε ελεύθερα την εντολή:

root@qemu:~$ dpkg -i package_name.deb

για τα 3 νέα αρχεία .deb που παρήγαγε η παραπάνω διαδικασία, και έτσι έχουμε εγκαταστήσει στο /boot/ directory του target συστήματος ένα image του πυρήνα και ένα ανανεωμένο initrd.

8.

Για να λειτουργήσει ορθά το συστήμα με τον νέο πυρήνα, κατεβάζουμε τα αρχεία αυτά στο host μηχάνημα και επανεκινούμε τον qemu δίνοντας τα ως ορισματα στα flags kernel και initrd.

Σε αυτό το σημείο έχει ολοκληρωθεί η εγκατάσταση του νέου πυρήνα με επιτυχία.

Ερωτήματα :

1.

Εκτελούμε :

:~$ uname -a

στο Qemu, πριν και μετά την εγκατάσταση του νέου πυρήνα, και παίρνουμε τα εξής αποτελέσματα :

Πριν :

```
root@debian-armhf:~# uname -a
Linux debian-armhf 3.2.0-4-vexpress #1 SMP Debian 3.2.51-1 armv7l GNU/Linux
```

Μετά:

```
root@debian-armhf:~# uname -a
Linux debian-armhf 3.16.84 #4 SMP Wed Feb 21 16:58:00 EET 2024 armv7l
GNU/Linux
```

Βλέπουμε πως η έκδοση του λειτουργικού έχει αλλάξει με επιτυχία και έχει το όνομα που θα περιμέναμε (Linux 3.16.84).

2.

Στο σημείο αυτό, θα χρειαστεί να δημιουργήσουμε το νέο αυτό system call για τον νέο πυρήνα. Αυτό θα υλοποιηθεί στον πυρήνα του host machine, linux-source-3.16, και στην συνέχεια θα εκτελέσουμε και πάλι τα βήματα 5-8. για να εγκαταστήσουμε εκ νέου τον πυρήνα ο οποίος περιέχει το νέο system call στο guest μηχάνημα.

1. Αρχίζουμε,λοιπόν, δημιουργώντας ένα νέο directory στο **~/ linux-source-3.16** με όνομα “hello”. Σε αυτό τον φάκελο, δημιουργούμε 2 προγράμματα, ένα με όνομα hello.c και ένα Makefile με τον αντίστοιχο τρόπο :

   Hello.c:

```c
#include <linux/kernel.h>

asmlinkage long sys_hello(void)
{
        printk(KERN_ALERT "Greeting from kernel and team 29!\n");
        return 0;
}
```

Makefile:

```
obj-y := hello.o
```

2. Προσθέτουμε το system call μας στο header file με όλα τα υπόλοιπα του πυρήνα. Συνεπώς, στο directory **~/ linux-source-3.16/include/linux/syscalls.h**,προσθέτουμε μαζί με τα υπόλοιπα sys_calls, το δικό μας, με την εξής γραμμή στο τέλος :

   asmlinkage long sys_hello(void);

3. Αλλάζουμε το αρχείο **Makefile** στο directory **~/ linux-source-3.16**, δίνοντας του το path για το system call μας. Αλλάζουμε επομένως την εντολή :

   core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block

   σε :

   core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello

4. Προσθέτουμε το system call μας στο system call table του πυρήνα μας, ο οποίος είναι βασισμένος σε arm αρχιτεκτονική. Επομένως, πηγαίνουμε στο directory **~/ linux-source-3.16/arch/arm/kernel/calls.S** και προσθέτουμε το δικό μας system call στο τέλος από τα υπόλοιπα γράφοντας τη γραμμή :

   /* 386 */ CALL(sys_hello)

   Ο αριθμός 386 προκύπτει από το γεγονός πως το αρχείο αρχικά χρησιμοποιούσε μέχρι και το 385 για τα υπόλοιπα system calls.

5. Τέλος, προσθέτουμε τον αριθμό αυτό στο αρχείο **~/ linux-source-3.16/arch/arm/include/uapi/asm/unistd.h** γράφοντας τη γραμμή :

   #define __NR_hello (__NR_SYSCALL_BASE+386).

Η υλοποίηση και η προσθήκη του νέου system call στον πυρήνα έχει ολοκληρωθεί. Το μόνο που μένει να κάνουμε είναι να επαναλάβουμε τα βήματα 5-8, για να ξαναχτίσουμε στο qemu τον νέο πυρήνα.

3.

Τέλος, είμαστε έτοιμοι να ελέγξουμε τη λειτουργία του νέου system call που δημιουργήσαμε. Δημιουργούμε στο qemu, λοιπόν, ένα πρόγραμμα test.c το οποίο εκτελεί το νέο syscall. Το μόνο που έχουμε να κάνουμε στον κώδικα είναι να εκτελέσουμε την εντολή syscall(sys_call_number) , όπου sys_call_number = 386 όπως δείξαμε παραπάνω.

Ακολουθεί ο κώδικας :

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#define SYS_new 386

int main(void) {
    printf("Testing System Call :\n");
    long sc = syscall(SYS_new);
    printf("Return Value = %ld\n", sc);
    return 0;
}
```

Κάνουμε compile με :

gcc test.c -o test

Εκτελούμε με :

./test

Και έχουμε ορθά, το εξής αποτέλεσμα :

```
root@debian-armhf:~# gcc test.c -o test
root@debian-armhf:~# ./test
Testing System Call :
[  209.397268] Greeting from kernel and team 29!
Return Value = 0
root@debian-armhf:~# _
```