ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Σχεδιασμός Ενσωματωμένων Συστημάτων
9ο Εξαμηνο ΗΜΜΥ

Συνεργάτες: Γεώργιος-Ταξιάρχης Γιαννιός, Α.Μ.: 03116156
Μπέτζελος Χρήστος, Α.Μ.: 03116067

5η Εργαστηριακή Άσκηση:
Cross-compiling προγραμμάτων για ARM αρχιτεκτονική

Προετοιμασία για την άσκηση

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

1. Αρχικά, δημιουργήσαμε ένα δεύτερο εικονικό μηχάνημα (Virtual Machine) στο QEMU. Προκειμένου να συμβεί αυτό, χρειάστηκε να κατεβάσουμε τα παρακάτω αρχεία (αντίστοιχα με τα αρχεία της εργαστηριακής άσκησης 3):
   - 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 3η 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.img3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_standard.qcow2
-append "root=/dev/mmcblk0p2" -net nic -net user,hostfwd=tcp:127.0.0.1:22223-:22
```

3. Επίσης, χρειάστηκε 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

Βήματα:

1. Αρχικά κατεβάσαμε τα απαραίτητα αρχεία για το κτίσιμο του toolchain από το αντίστοιχο github repository. 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$ ./bootstrap
```

3. Στη συνέχεια, θα πρέπει να δημιουργήσουμε 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 κατεβαζει για να χτίσει τον cross-compiler.

4. Εκτελούμε την παρακάτω εντολή για να κάνουμε configure την εγκατάσταση του crosstool-ng:

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

Κατά τη διάρκεια της εκτέλεσης αυτής της εντολής, πολλά πακέτα έλειπαν. Τα εγκαταστήσαμε, αναζητώντας το κάθε πακέτο σε κάποια μηχανή αναζήτησης, και στη συνέχεια ξαναεκτελέσαμε την παραπάνω εντολή.

5. Εκτελούμε την εντολή make και makeinstall:

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

6. Μέχρι εδώ έχει εγκατασταθεί το crosstool-ng. Πηγαίνουμε στο installation path *$HOME/crosstool/bin*

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

Σε αυτό το φάκελο κάνουμε build τον cross 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 επιλέξαμε 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 αλλάξουμε με γραφικό τρόπο κάποια χαρακτηριστικά του preconfigured συνδυασμού target machine, όπως για παράδειγμα ποια βιβλιοθήκη της C θα χρησιμοποιήσουμε, εκτελούμε την εντολή:

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

10. Τέλος αφού έχουμε παραμετροποιήσει τον crosscompiler είμαστε έτοιμοι να τον κάνουμε build. Το χτίσιμο του cross compiler είναι μία σχετικά χρονοβόρα διαδικασία. Εκτελούμε:

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

11. Αφού όλα έχουν πάει καλά, δημιουργήθηκε ο φάκελος `$HOME/x-tools/arm-cortexa9_neon-linux-gnueabihf` όπου μέσα στον υποφάκελο bin περιέχει τα εκτελέσιμα αρχεία του cross compiler σας

Παρατήρηση:

Το crosstool-ng δεν μπορούσε να κατεβάσει κάποια αρχεία που χρειάζονταν, οπότε τα κατεβάσαμε εμείς manually και τα τοποθετήσαμε στο φάκελο που υποδείκνυε το build.log αρχείο.

Εγκατάσταση pre-compiled building toolchain linaro

Εκτός από τη χρήση του custom compiler toolchain, χρησιμοποιήσαμε 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 από την παρακάτω διεύθυνση:

```
~$mkdir~/linaro && cd ~/linaro
:~/linaro                                                    $wget
https://releases.linaro.org/archive/14.04/components/toolchain/binaries/
gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz2
```

2. Ανοίγοντας το parent directory του tarball που κατεβάσαμε (https://releases.linaro.org/archive/14.04/components/toolchain/binaries/), παρατηρούμε στο κάτω μέρος της σελίδας ότι τα binaries του cross compiler συμπεριλαμβάνουν τα παρακάτω στοιχεία:

- Linaro GCC 4.8 2014.04
- A statically linked gdbserver
- Linaro Newlib 2.1 2014.02
- A system root
- Linaro Binutils 2.24.0 2014.04
- Manuals under share/doc/
- Linaro GDB 7.6.1 2013.10
- The system root contains the basic header files and libraries to link your programs against.

3. Κάνουμε extract τα αρχεία που κατεβάσαμε.

```
:~/linaro$ tar -xvf gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz24
```

4. Τα binaries του crosscompiler θα πρέπει να βρίσκονται στο πακέτο που κατεβάσαμε στον φάκελο:

```
$HOME/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin
```

Άσκηση 1

1. Η **armel** (arm EABI little-endian) αρχιτεκτονική υποστηρίζει το σύνολο εντολών ARMv4. Αυτή η αρχιτεκτονική χειρίζεται τον υπολογισμό αριθμών κινητής υποδιαστολής σε λειτουργία συμβατότητας που επιβραδύνει την απόδοση, αλλά επιτρέπει τη συμβατότητα με κώδικα γραμμένο για επεξεργαστές χωρίς μονάδες κινητής υποδιαστολής. Έτσι μπορεί να χρησιμοποιηθεί η αρχιτεκτονική του armel για να δημιουργηθούν συστήματα υψηλής συμβατότητας.

Η **armhf** (arm hard-float) αρχιτεκτονική υποστηρίζει την πλατφόρμα ARMv7, και επιπλέον, προσθέτει άμεση hardware υποστήριξη κινητών υποδιαστολών. Αυτό σημαίνει ότι η αρχιτεκτονική του armhf είναι ταχύτερη από αυτήν του armel, αλλά στερείται της συμβατότητας με τις παλαιές αρχιτεκτονικές.

Architecture	Debian Designation	Subarchitecture	Flavor
Intel x86-based	i386	default x86 machines	default
		Xen PV domains only	xen
AMD64 & Intel 64	amd64		
ARM	armel	Marvell Kirkwood and Orion	marvell
ARM with hardware FPU	armhf	multiplatform	armmp
64bit ARM	arm64		

2. Χρησιμοποιήσαμε την “arm-cortexa9_neon-linux-gnueabihf”, γιατί το guest μηχάνημα έχει αυτή την αρχιτεκτονική. Σε αντίθετη περίπτωση, αν προσπαθούσαμε να τρέξουμε ένα cross-compiled εκτελέσιμο διαφορετικής αρχιτεκτονικής, το εκτελέσιμο θα μας έβγαζε σφάλμα, καθώς αναμένεται μια τελείως διαφορετική αρχιτεκτονική όσον αφορά το Instruction Set Architecture του συστήματος.

3. Η βιβλιοθήκη της C που χρησιμοποιήσαμε είναι η **glibc**, καθώς ταιριάζει περισσότερο στις ανάγκες μας. Υπήρχαν δυνατότητες χρήσης βιβλιοθηκών που είναι μικρότερες, αλλά αυτές ή δεν υποστηρίζουν ARM ή δεν υποστηρίζουν το distribution Debian, οπότε δεν μπορούμε να τις χρησιμοποιήσουμε.

Αυτό φαίνεται χρησιμοποιώντας την εντολή:

```
$ ldd -v ~/x-tools/arm-cortexa9_neon-
linux-gnueabihf/bin/arm-cortexa9_neon-linux-gnueabihf-gcc
```

```
linux-vdso.so.1 (0x00007fff849f7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4eeac15000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4eeb006000)

Version information:
/home/user/x-tools/arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon-linux-gnueabihf-gcc:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
        libc.so.6 (GLIBC_2.3) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.9) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.14) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.4) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.11) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.3.4) => /lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
```

4. Χρησιμοποιώντας τον cross compiler που παρήχθη από τον crosstool-ng κάνουμε compile τον κώδικα `phods.c` με flags `-O0 -Wall -o phods_crosstool.out` από το 2ο ερώτημα της 1ης άσκησης (τον απλό κώδικα phods μαζί με την συνάρτηση gettimeofday()).

```
sudo ./arm-cortexa9_neon-linux-gnueabihf-gcc phods.c -O0 -Wall -o
phods_crosstool.out
```

Τρέχουμε στο τοπικό μηχάνημα τις εντολές:

```
:~$ 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
```

```
:~$ 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:               0x10460
  Start of program headers:          52 (bytes into file)
  Start of section headers:          16740 (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:         37
  Section header string table index: 36
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
```

Οι εντολές αυτές μας δίνουν πληροφορίες, από τα headers του binary αρχείου, σχετικά με τον τυπο του, το ABI, την targeted αρχιτεκτονική (Machine) κ.α. Πιο ειδικά, το πρώτο πράγμα που παρατηρούμε είναι ότι η αρχιτεκτονική πάνω στην οποία τρέχει το αρχείο είναι ARM. Ακόμα, λειτουργεί με dynamic linking και το αρχείο της βιβλιοθήκης που καλεί είναι το /lib/ld-linux-armhf.so.3. Εχουμε ακόμα πολλές πληροφορίες για τις CPU για τις οποίες είναι φτιαγμένο αυτό το αρχείο, καθώς και για το ABI, ώστε να γνωρίζει το λειτουργικό λεπτομέρειες για την διαχείριση πράξεων κλπ.

5. Χρησιμοποιώντας τον cross compiler που κατεβάσαμε από το site της linaro, κάνουμε compile τον ίδιο κώδικα με το ερώτημα 3.

```
~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin$
./arm-linux-gnueabihf-gcc-4.8.3 phods.c -O0 -Wall -o phods_linaro.out
```

- arm-cortexa9: 18.2 kB (18,220 bytes)
- linaro: 8.4 kB (8,355 bytes)

Βλέπουμε, λοιπόν, πως το εκτελέσιμο *phods_crosstool.out* έχει παραπάνω από διπλάσιο μέγεθος σε σχέση με το εκτελέσιμο *phods_linaro.out*. Αυτό συμβαίνει διότι ο custom cross-compiler κάνει χρήση 64-bit glibc ενώ ο linaro 32-bit. Αυτό φαίνεται και από την παρακάτω εντολή.

```
ldd -v arm-linux-gnueabihf-gcc-4.8.3
```

6. Το εκτελέσιμο που παράχθηκε από τον linaro (*phods_linaro.out*), του ερωτήματος 5, εκτελείται σωστά στο target μηχάνημα διότι, ένα 64-bit σύστημα μπορεί να εκτελέσει 32-bit εκτελέσιμα. Για να πραγματοποιηθεί αυτό εκτελέστηκαν τα παρακάτω:

Enable i386 package installation support:

```
sudo dpkg --add-architecture i386
sudo apt-get update
```

Install packages need to build source code on the system:

```
sudo apt-get install git build-essential fakeroot
```

gcc-multilib is the package which will enable running 32bit (x86) binaries on a 64bit (amd64/x86_64) system.

```
sudo apt-get install gcc-multilib
sudo apt-get install zlisudo apt-get install git build-essential fakeroot
```

7. Εκτελέσαμε τα ερωτήματα 4 και 5 με επιπλέον flag `-static`. Το flag που προσθέσαμε ζητάει από τον εκάστοτε compiler να κάνει στατικό linking της αντίστοιχης βιβλιοθήκης της C του κάθε compiler. Τα μεγέθη των δύο αρχείων φαίνονται παρακάτω:

- arm-cortexa9: 2.9 MB (2,923,704 bytes)
- linaro: 507.9 kB (507,938 bytes)

Παρατηρούμε διαφορά στο μέγεθος, καθώς το static linking έχει ενσωματωμένες όλες τις 64-bit βιβλιοθήκες της C που κάνει reference το πρόγραμμα *phods*. Δηλαδή, οι βιβλιοθήκες αποτελούν μέρος των τελικών εκτελέσιμων πράγμα που αυξάνει κατακόρυφα το μέγεθος τους.

Όπως φαίνεται και από τα παραπάνω, μπορεί η επιλογή του cross-compiler να επιδρά στο μέγεθος του εκτέσιμου (όπως σχολιάστηκε και πριν), αλλά τον πιο σημαντικό ρόλο τον έχει η επιλογή ανάμεσα σε δυναμική σύνδεση των βιβλιοθηκών και σε στατική.

8. Προσθέτουμε μία δική μας συνάρτηση στη `mlab_foo()` στη glibc και δημιουργούμε έναν cross-compiler 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. Αν εκτελέσουμε το `my_foo.out` στο host μηχάνημα, θα παρατηρήσουμε ότι **δεν μπορεί** να εκτελεστεί, διότι το εκτελέσιμο έχει φτιαχτεί για άλλη αρχιτεκτονική (για την αρχιτεκτονική του target μηχανήματος).

B. Αν εκτελέσουμε το `my_foo.out` στο target μηχάνημα, θα παρατηρήσουμε ότι **δεν μπορεί** να εκτελεστεί διότι το target μηχάνημα δεν έχει την νέα βιβλιοθήκη για να την κάνει dynamically link.

C. Προσθέτουμε το flag `-static` και κάνουμε compile ξανά το αρχείο `my_foo.c`. Αν εκτελέσουμε το `my_foo.out` στο target μηχάνημα, θα παρατηρήσουμε ότι τώρα **μπορεί** να εκτελεστεί διότι, πλέον το εκτελέσιμο έχει ενσωματωμένο το binary της νέας βιβλιοθήκης (αφού κάναμε compile με το flag -static).

Ασκηση 2

1. Εκτελούμε στο Qemu

```
:~$ uname -a
```

Πριν την εγκατάσταση νέου πυρήνα:

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

Αφότου έχουμε εγκαταστήσει τον νέο πυρήνα:

```
root@debian-armhf:~# uname -a
Linux debian-armhf 3.16.84 #2 SMP Thu Jan 7 12:47:16 EET 2021 armv7l GNU/Linux
```

Αυτό που παρατηρούμε να αλλάζει ουσιαστικά είναι η έκδοση του λειτουργικού, η οποία πλέον είναι 3.16.84, ενώ πριν ήταν 3.2.0.4.

2. Προσθέσαμε στον πυρήνα του linux ένα καινούριο system call που χρησιμοποιεί την συνάρτηση *printk* για να εκτυπώνει στο log του πυρήνα την φράση "*Greeting from kernel and team no %d*" μαζί με όνομα της ομάδας μας.

Οι αλλαγές που κάναμε στον πηγαίο κώδικα του πυρήνα είναι:

- Αρχικά φτιάχνουμε ένα φάκελο ***/new*** στο *linux-source-3.16* directory με δύο αρχεία. Ένα ***new.c*** και ένα ***Makefile*** που θα το μετατρέψει σε object file.

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

asmlinkage long sys_new(void)
{
        printk(KERN_ALERT "Greeting from kernel and team 20!\n");
        return 0;
}
```

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

- Στο αρχείο */include/linux/syscalls.h* προσθέτουμε

```c
asmlinkage long sys_new(void);
```

- Μετά προσθέτουμε στο αρχείο */arch/arm/include/asm/unistd.h*

```c
#define __NR_sys_new (__NR_SYSCALL_BASE+378)
```

- Έπειτα προσθέτουμε στο αρχείο */arch/arm/kernel/calls.S*

```
/* 385 */    CALL(sys_memfd_create)
        CALL(sys_new)
```

- Τέλος προσθέτουμε τον φάκελο του system call μας στο Makefile του kernel στον κανόνα για τα *core-y*, τροποποιώντας την παρακάτω μία γραμμή στο αρχείο /Makefile.

  από

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

  σε

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

- Κάνουμε **compile** και φορτώνουμε το νέο πυρήνα με την νέα κλήση συστήματος.

3. Τέλος, γράψαμε ένα πρόγραμμα σε γλώσσα C, το οποίο κάνει χρήση του system call που προσθέσαμε. Παρακάτω, παρατίθεται ο κώδικας ***test.c*** για τον έλεγχο της ορθής λειτουργίας του νέου system call.

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#define SYS_new 386
int main(void) {
      printf("Invoking system call.\n");
      long ret = syscall(SYS_new);
      printf("Syscall returned %ld.\n", ret);
      return 0;
}
```

```
root@debian-armhf:~# gcc test.c -o test
root@debian-armhf:~# ./test
Invoking system call.
[  433.302261] Greeting from kernel and team 20!
Syscall returned 0.
```