

**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΕΡΓΑΣΤΗΡΙΟ ΗΛΕΚΤΡΟΝΙΚΗΣ

**Σχεδιασμός Ενσωματωμένων Συστημάτων**

9ο εξάμηνο

5η Εργαστηριακή Άσκηση:

**Cross-compiling προγραμμάτων για ARM αρχιτεκτονική**

ΑΜ: 03116112

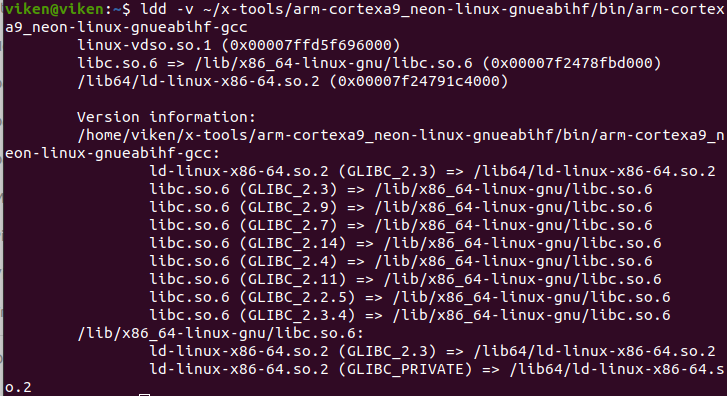
Ονοματεπώνυμο: Γεζεκελιάν Βικέν

**Άσκηση 1**

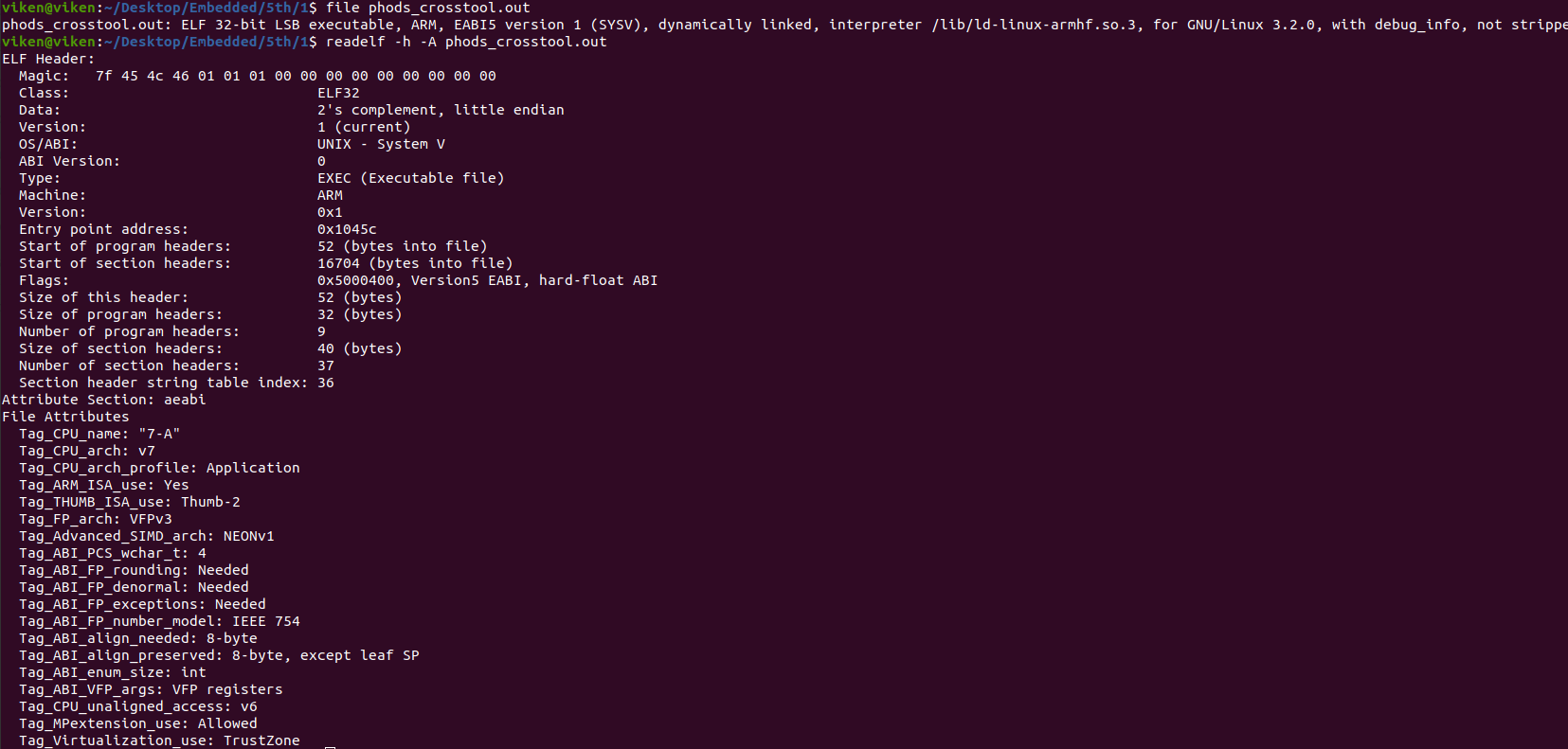
1. Η διαφορά μεταξύ των δύο images, έγκειται στην χρήση των floating point units από μέρους του armhf. Και στις δύο περιπτώσεις χρησιμοποιείται 32-bit αρχιτεκτονική, με το armel image να είναι πρακτικά ένα παλαιότερο port, το οποίο δεν έχει τη δυνατότητα να χρησιμοποιεί τα FPU που περιέχουν οι περισσότεροι σημερινοί επεξεργαστές. Ο armhf αντιθέτως τα χρησιμοποιεί παρέχοντας βελτιωμένες επιδόσεις. Τα el και hf στα ονόματα των επεξεργαστών δηλώνουν little-endian και hard-float αντίστοιχα

2. Ο λόγος που επιλέξαμε τη συγκεκριμένη αρχιτεκτονική αφορά το vm που θα χρησιμοποιηθεί για τη 2η άσκηση, που κάνει emulate arm αρχιτεκτονική. Σε περίπτωση που επιλέγαμε κάποια μη arm αρχιτεκτονική θα εμφανιζόταν exec format error.

3. Όπως φαίνεται και παρακάτω, με την εκτέλεση της εντολής “ldd -v ~/x-tools/arm-cortexa9\_neon-linux-gnueabihf/bin/arm-cortexa9\_neon-linux-gnueabihf-gcc” βλέπουμε πως χρησιμοποιείται η glibc (GNU C lib), που είναι η default βιβλιοθήκη.



4. ~/x-tools/arm-cortexa9\_neon-linux-gnueabihf/bin/arm-cortexa9\_neon-linux-gnueabihf-gcc phods.c -O0 -Wall -o phods\_crosstool.out



Όπως φαίνεται και στην παραπάνω εικόνα, οι δύο εντολές αυτές μας δίνουν πληροφορίες σχετικά με το αρχείο που δώσαμε ως είσοδο χρησιμοποιώντας τα headers του. Συγκεκριμένα, από την εντολή file συμπεραίνουμε πως το αρχείο πρόκειται για εκτελέσιμο, καθώς και την αρχιτεκτονική του, le ή be, dynamically/statically linked, interpreter path, kernel version και αν είναι stripped. Η readelf μας δίνει ακόμη περισσότερες πληροφορίες σχετικά με το πρόγραμμα, οι οποίες φαίνονται αναλυτικά τόσο στην παραπάνω εικόνα, όσο και στη man page.

5. ~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04\_linux/bin/arm-linux-gnueabihf-gcc phods.c -O0 -Wall -o phods\_linaro.out

Χρησιμοποιώντας τον cross compiler που δημιουργήσαμε, το παραγόμενο εκτελέσιμο έχει μέγεθος 18 KB, ενώ με τη χρήση του compiler του linaro έχει μέγεθος 9 KB. Βλέπουμε λοιπόν πως το εκτελέσιμο που παράγει ο cross compiler της linaro έχει πολύ μικρότερο μέγεθος. Το γεγονός αυτό οφείλεται εν μέρει στις βελτιστοποιήσεις που σίγουρα έχουν γίνει στον compiler που κατεβάσαμε, αλλά οφείλονται σε μεγάλο βαθμό και στο γεγονός πως ο cross compiler μας χρησιμοποιεί 64 bit βιβλιοθήκες με 32 bit support, σε αντίθεση με τον compiler της linaro που χρησιμοποιεί απευθείας τις 32 bit βιβλιοθήκες που απαιτούνται

6. Επειδή τα 64 bit συστήματα υποστηρίζουν την εκτέλεση 32 bit προγραμμάτων.

7. ~/x-tools/arm-cortexa9\_neon-linux-gnueabihf/bin/arm-cortexa9\_neon-linux-gnueabihf-gcc phods.c -O0 -Wall -static -o phods\_crosstool\_static.out

~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04\_linux/bin/arm-linux-gnueabihf-gcc phods.c -O0 -Wall -static -o phods\_linaro\_static.out

Παρατηρούμε πως τα μεγέθη των παραγόμενων εκτελέσιμων είναι τραγικά μεγαλύτερα, συγκεκριμένα 2.855ΚΒ και 496ΚΒ για τον cross compiler μας και τον compiler της linaro αντίστοιχα. Ο λόγος είναι πως πλέον στα αρχεία αυτά εμπεριέχεται ο κώδικας των προηγουμένως δυναμικά συνδεδεμένων βιβλιοθηκών. Επίσης, παρατηρούμε πως η διαφορά μεγέθους μεταξύ των δύο είναι πλέον ακόμη μεγαλύτερη, πιθανώς λόγω του πλήθους των βιβλιοθηκών που πρέπει να συμπεριληφθούν σε κάθε περίπτωση.

8.

A. Στο host μηχάνημα ο κώδικας δεν θα τρέξει, αφού είναι παραγόμενος από cross compilation για διαφορετική αρχιτεκτονική.

B. Το πρόγραμμα θα τρέξει στο target μηχάνημα μόνο αφότου προσθέσουμε την καινούργια βιβλιοθήκη της glibc σε αυτό. Σε αντίθετη περίπτωση δεν θα γίνει ποτέ το dynamic linking και δεν θα τρέξει.

C. Το πρόγραμμα θα τρέξει κανονικά, αφού οι απαραίτητες συναρτήσεις θα συμπεριλαμβάνονται εντός του εκτελέσιμου.

Στη συγκεκριμένη άσκηση τα προβλήματα που αντιμετωπίσαμε αφορούσαν κατά κύριο λόγο τις βιβλιοθήκες που δεν υπήρχαν εγκατεστημένες στο σύστημά μας και αφορούν το 32-bit support. Παρ’ όλα αυτά, η εγκατάσταση τους ήταν εύκολη, αναζητώντας κάθε φορά online ποια βιβλιοθήκη περιλαμβάνει το πακέτο που λείπει και εγκαθιστώντας το.

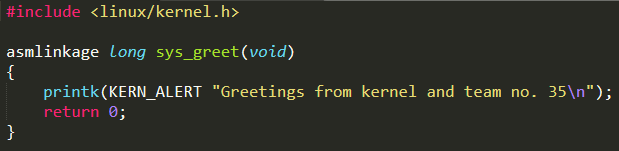
**Άσκηση 2**

1. Παρακάτω, φαίνονται τα αποτελέσματα της εντολής uname -a, πριν και μετά την εγκατάσταση του νέου πυρήνα:



Βλέπουμε πως ο νέος πυρήνας έχει εγκατασταθεί σωστά και χρησιμοποιείται τη δεύτερη φορά που ανοίγουμε το vm.

2. Για την υλοποίηση του ζητούμενου syscall, αρχικά δημιουργήσαμε τον κώδικα που βρίσκεται και εντός του αρχείου greet.c με το αντίστοιχο makefile του.



Έπειτα, χρειάστηκε να μεταβούμε στον φάκελο linux-source-3.16, όπου δημιουργήσαμε τον φάκελο greet, εντός του οποίου τοποθετήσαμε τον παραπάνω κώδικα και makefile.

Στη συνέχεια έπρεπε να κάνουμε register το syscall στο header file του πυρήνα, το οποίο κάναμε προσθέτοντας την γραμμή :

asmlinkage long sys\_greet(void);

στο αρχείο linux-source-3.16/include/linux/syscalls.h

Έπειτα, μεταβήκαμε στο αρχείο linux-source-3.16/arch/arm/kernel/calls.S, όπου στο τέλος του, αφού είδαμε πως το τελευταίο syscall είναι το:

/\* 385 \*/ CALL(sys\_memfd\_create)

προσθέσαμε τη γραμμή:

/\* 386 \*/ CALL(sys\_greet)

Ομοίως, προσθέσαμε την γραμμή:

#define \_\_NR\_sys\_greet (\_\_NR\_SYSCALL\_BASE+386)

στο αρχείο linux-source-3.16/arch/arm/include/asm/unistd.h

Στη συνέχεια, προσθέσαμε την γραμμή:

386 i386 sys\_greet sys\_greet

Στο αρχείο linux-source-3.16/arch/x86/syscalls/syscall\_32.tbl

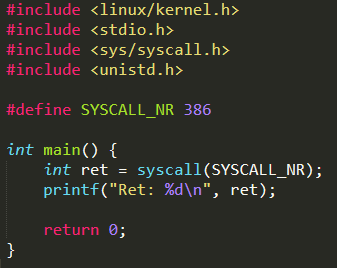
Τέλος, στο Makefile εντός του φακέλου linux-source-3.16 συμπεριλάβαμε τον φάκελο /greet:

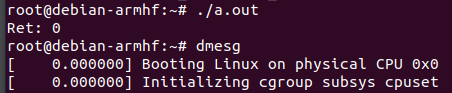
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ greet/

Το μόνο που μένει έπειτα είναι να μεταγλωττίσουμε τον πυρήνα με τις καινούργιες ρυθμίσεις(που έγινε όπως αναφέρεται και στην εκφώνηση χωρίς προβλήματα) και να χρησιμοποιήσουμε το καινούργιο image για να ανοίξουμε το vm.

3. Για το ερώτημα αυτό, χρειάζεται να μεταφέρουμε απλώς το αρχείο test.c που καλεί το system call μας στο vm και το κάνουμε compile. Όπως βλέπουμε, επιστρέφεται το 0, γεγονός που σημαίνει πως έτρεξε σωστά, ενώ με dmesg βλέπουμε και το μήνυμα να εκτυπώνεται στο log του πυρήνα.

Ο κώδικας του test.c καθώς και το log του πυρήνα φαίνονται παρακάτω:





(Υπήρξαν διάφορα άλλα στο log του πυρήνα που δεν έχουν σημασία)

