Σχεδιασμός Ενσωματωμένων Συστημάτων

5η Εργαστηριακή Άσκηση

Cross-compiling προγραμμάτων για ARM αρχιτεκτονική

Νικολέτα-Μαρκέλα Ηλιακοπούλου
03116111

Φοίβος-Ευστράτιος Καλεμκερής
03116010

1 Εγκατάσταση cross-compiler building toolchain crosstool-ng

Για την εγκατάσταση του crosstool-ng ακολουθήσαμε τα βήματα που παρουσιάζονται στην εκφώνηση, χωρίς να αντιμετωπίσουμε προβλήματα στα περισσότερα απ' αυτά. Ειδικότερα:

1. Κατεβάσαμε τα απαραίτητα αρχεία για το χτίσιμο του toolchain από το αντίστοιχο github repository με την ακόλουθη εντολή:

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

2. Μπήκαμε στο φάκελο crosstool-ng, και εκτελέσαμε:

   ```
   :~$ cd crosstool-ng && ./bootstrap
   ```

3. Δημιουργήσαμε 2 φακέλους στο HOME directory:

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

4. Για να κάνουμε configure την εγκατάσταση του crosstool-ng εκτελέσαμε:

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

   Στο σημείο αυτό εμφανίστηκαν πολλά μηνύματα για πακέτα που έλειπαν. Χρησιμοποιήσαμε τον package manager για την εγκατάσταση καθενός εξ αυτών:

   - (flex) -> sudo apt-get install flex
   - (makeinfo) -> sudo apt-get install texinfo
   - (help2man) -> sudo apt-get install help2man
   - (GNU awk) -> sudo apt-get install gawk
   - (libtool) -> sudo apt install libtool-bin libtool-doc
   - (curses) -> sudo apt-get install ncurses-dev

5. Εκτελέσαμε make και make install:

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

   Το make αρχικά αποτυγχάνει λόγω απουσίας του yacc. Το αντιμετωπίσαμε κατεβάζοντας τα παρακάτω και έπειτα επαναλαμβάνοντάς το.

   - sudo apt-get install bison -y
   - sudo apt-get install byacc -y

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

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

   Σε αυτό το φάκελο κάναμε build τον cross compiler.

7. Εκτελέσαμε την εντολή

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

   για να παραμετροποιήσουμε το crosstool-ng για την αρχιτεκτονική arm-cortexa9_neon-linux-gnueabihf

8. Δεν αλλάζουμε κάτι άλλο στις ρυθμίσεις, αφού θα χρησιμοποιήσουμε τη βιβλιοθήκη της C glibc, η οποία είναι ήδη επιλεγμένη. Κάναμε build εκτελώντας:

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

   Μετά το τέλος του build έχει δημιουργηθεί ο φάκελος $HOME/x-tools/arm-cortexa9_neon-linux-gnueabihf, όπου μέσα στον υποφάκελο bin περιέχει τα εκτελέσιμα αρχεία του cross-compiler.

2 Εγκατάσταση pre-compiled cross compiler

Κατά το κατέβασμα του linaro δεν αντιμετωπίσαμε κανένα πρόβλημα ακολουθώντας τα βήματα της εκφώνησης:

1. Κατεβάσαμε 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. Κάναμε extract τα αρχεία που κατεβάσαμε:

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

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.

3 Άσκηση 1

Προτού προχωρήσουμε με την εκτέλεση της άσκησης, στήνουμε τον qemu, έτσι ώστε να χρησιμοποιεί το καινούριο image (armhf), προκειμένου να αποφύγουμε προβλήματα συμβατότητας.

1. Στην εργαστηριακή άσκηση 3 είχαμε χρησιμοποιήσει για τον qemu το image debian_wheezy_armel, ενώ αυτή τη φορά προτιμήσαμε το debian_wheezy_armhf. Τα δύο images αποτελούν απλώς ports του Debian OS για διαφορετικές αρχιτεκτονικές. Ειδικότερα, το armel (arm little-endian) είναι το παλαιότερο από τα υποστηριζόμενα ports του Debian ARM και υποστηρίζει little-endian 32-bit αρχιτεκτονικές, συμβατές με το v5te instruction set. Το armhf (arm hard-float) από την άλλη εκμεταλλεύεται το floating-point unit (FPU) που βρίσκεται ενσωματωμένο σε πολλά σύγχρονα 32-bit ARM boards και σε αντίθεση με το armel, παρέχει υποστήριξη για εκτέλεση floating-point operations στο hardware. Προϋποθέτει, ωστόσο, σύστημα με τουλάχιστον ARMv7 CPU με Thumb-2 και VFPv3-D16 υποστήριξη floating point.

2. Χρησιμοποιούμε την αρχιτεκτονική arm-cortexa9_neon-linux-gnueabihf, αφού αυτή είναι συμβατή με το σύστημα που έχουμε στήσει στον qemu. Ειδικότερα, όπως μαρτυρά και το όνομα, υποστηρίζει εκτέλεση floating-point operations στο hardware (hf = hard-float) και παρέχει συμβατότητα με το EABI που χρησιμοποιεί και το armhf. Επιπλέον, όπως πληροφορούμαστε από το site της arm, ο Cortex A9 χρησιμοποιεί τον 32-bit πυρήνα Armv7-A, πληρώντας ταυτόχρονα τις ελάχιστες απαιτήσεις του armhf για Thumb-2 και VFPv3-D16 υποστήριξη floating point. Σε περίπτωση που χρησιμοποιούσαμε διαφορετική αρχιτεκτονική θα προέκυπταν προβλήματα συμβατότητας ως προς την υποστηριζόμενη ISA (διαφορετικός αριθμός bit αρχιτεκτονικής και instruction set, μη υποστήριξη εκτέλεσης FLOPs στο hardware κλπ).

3. Στο βήμα 9 χρησιμοποιήσαμε τη βιβλιοθήκη glibc. Η επιλογή μας αυτή βασίστηκε κυρίως στο γεγονός ότι η συγκεκριμένη βιβλιοθήκη υποστηρίζεται σίγουρα από τη Debian και είναι συμβατή με την ARMv7 αρχιτεκτονική, ενώ προτείνεται στο site της Debian ως μέρος του toolchain για cross-compiling. Αυτό δε σημαίνει απαραίτητα ότι αποκλείσαμε όλες τις υπόλοιπες παρεχόμενες βιβλιοθήκες (bionic, uClibc, musl), οι οποίες καταλαμβάνουν πολύ μικρότερο χώρο και ενδέχεται με ελάχιστες προσαρμογές να λειτουργούν κανονικά στο σύστημά μας, η glibc αποτελεί απλώς την "ασφαλή" επιλογή. Στην έξοδο της ακόλουθης εντολής μπορεί να φανεί η χρησιμοποιούμενη βιβλιοθήκη:

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

linux-vdso.so.1 (0x00007ffcc03fb000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f864169a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f86418a4000)
```

```
Version information:
/home/phoevos/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_2.7) => /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 παρήχθει από τον crosstool-ng κάνουμε compile τον κώδικα phods_initial.c με flags -O0 -Wall -o phods_crosstool.out

```
$ ~/x-tools/arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon-
    linux-gnueabihf-gcc -O0 -Wall -o phods_crosstool.out phods_initial.c
```

Ύστερα, στο τοπικό μηχάνημα τρέχουμε την ακόλουθη εντολή:

```
:~$ 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
```

Όπως φαίνεται από την παραπάνω έξοδο, η εντολή παρέχει πληροφορίες για το είδος του παραχθέντος αρχείου. Ειδικότερα, πληροφορούμαστε ότι αποτελεί εκτελέσιμο (ELF) για 32-bit ARM αρχιτεκτονική με υποστήριξη στο EABI5. Η πληροφορία αυτή είναι απαραίτητη προκειμένου να διασφαλίσουμε ότι το εκτελέσιμο που παράξαμε είναι συμβατό με την αρχιτεκτονική για την οποία προορίζεται. Μαθαίνουμε, επιπλέον, ότι είναι dynamically linked, ενώ λαμβάνουμε και το path του interpreter.
Στη συνέχεια, τρέχουμε την επόμενη εντολή:

```
:~$ 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:          16732 (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
  Tag_Virtualization_use: TrustZone
```

Η παραπάνω εντολή, όπως φαίνεται, δίνει αναλυτικότερες πληροφορίες σε σχέση με το εκτελέσιμο, οι οποίες προέρχονται από τα headers του binary αρχείου. Οι πιο σημαντικές εξ αυτών αφορούν την αρχιτεκτονική στόχο και έχουν να κάνουν με τη ISA, την υποστήριξη hard-float ABI και τη χρήση VFP αναφορικά με την εκτέλεση FLOPs. Όπως αναφέραμε ήδη, τα χαρακτηριστικά αυτά παρέχονται από την αρχιτεκτονική που έχουμε επιλέξει για το guest μηχάνημα (armhf).

5. Χρησιμοποιώντας τον cross compiler που κατεβάσαμε από το site της linaro κάνουμε, ακολούθως, compile τον ίδιο κώδικα, με την εντολή:

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

Για να λειτουργήσει το προηγούμενο, χρειάστηκε το παρακάτω πακέτο:

```
$ sudo apt-get install lib32z1-dev
```

Στη συνέχεια, με την εντολή ls -lh, μπορούμε να μάθουμε το μέγεθος των εκτελέσιμων:

```
-rwxrwxr-x 1 phoevos phoevos  18K I    5 16:35 phods_crosstool.out
-rwxrwxr-x 1 phoevos phoevos 8,2K I    5 19:05 phods_linaro.out
```

Όπως πληροφορούμαστε, το εκτελέσιμο που παράγεται από τον custom cross compiler έχει μέγεθος 18KB, ενώ, αυτό που παράγει ο compiler του linaro, 8.2KB. Εκτελώντας την εντολή

```
$ ldd -v ~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin/arm-
    linux-gnueabihf-gcc
```

πληροφορούμαστε ότι ο compiler του linaro χρησιμοποιεί επίσης τη glibc, αλλά την έκδοση των 32 bit. Είναι, σύνεπως, απολύτως αναμενόμενο το εκτελέσιμο που παράγεται να έχει πολύ μικρότερο μέγεθος.

6. Το πρόγραμμα του προηγούμενο ερωτήματος εκτελείται κανονικά στο target μηχάνημα, αφού ένα 64-bit σύστημα μπορεί να εκτελέσει 32-bit εκτελέσιμα.

7. Κάνουμε ξανά compile το phods_initial.c και με τους δύο cross compilers, με επιπλέον flag αυτή τη φορά -static. Το flag που προσθέσαμε ζητάει από τον εκάστοτε compiler να κάνει στατικό linking της αντίστοιχης βιβλιοθήκης της C. Στη συνέχεια, επαναλαμβάνοντας την εντολή ls -lh, παίρνουμε την ακόλουθη έξοδο:

```
-rwxrwxr-x 1 phoevos phoevos 2,8M I    5 19:26 phods_crosstool.out
-rwxrwxr-x 1 phoevos phoevos 497K I    5 19:26 phods_linaro.out
```

   Όπως βλέπουμε, το μέγεθος και των δύο εκτελέσιμων είναι κατά πολύ αυξημένο σε σχέση με την προηγούμενη μεταγλώττιση. Αυτό συμβαίνει, διότι, κατά τη στατική σύνδεση των βιβλιοθηκών, αυτές συμπεριλαμβάνονται στο τελικό εκτελέσιμο (binary αρχείο) που παράγεται.

8. Εξετάζουμε, τέλος, το ακόλουθο υποθετικό σενάριο: Προσθέτουμε μία δική μας συνάρτηση 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

   (α′) Αν επιχειρήσουμε να εκτελέσουμε το my_foo.out στο host μηχάνημα, αυτό δε θα τρέξει, αφού προορίζεται για διαφορετική αρχιτεκτονική (cross-compiled).

   (β′) Αν εκτελέσουμε το my_foo.out στο target μηχάνημα, η εκτέλεση θα αποτύχει και πάλι, αφού σ' αυτό υπάρχει μόνο η προηγούμενη έκδοση της glibc που δεν περιλαμβάνει τη συνάρτηση mlab_foo().

   (γ′) Αν προσθέσουμε το flag -static κατά τη μεταγλώττιση του my_foo.c στο host μηχάνημα και, στη συνέχεια, επιχειρήσουμε να εκτελέσουμε το my_foo.out στο target μηχάνημα, αυτό θα τρέξει κανονικά, αφού το linking με τη βιβλιοθήκη έχει γίνει στατικά κατά τη μεταγλώττιση (όχι κατά το run-time όπως θα συνέβαινε για dynamic linking), οπότε αυτή έχει συμπεριληφθεί στο εκτελέσιμο.

4 Άσκηση 2

1. Σ' αυτό το μέρος της άσκησης θα χρησιμοποιήσουμε τον cross compiler που χτίσαμε με το crosstool-ng, προκειμένου να χτίσουμε έναν νέο πυρήνα για το Debian OS.

2. Κατεβάσαμε, αρχικά, το directory /boot από το debian του qemu στο host μηχάνημα, ώστε να είμαστε σε θέση στη συνέχεια να συγκρίνουμε τα υπάρχοντα αρχεία του linux image και του initrd με αυτά που θα προκύψουν στο τέλος της διαδικασίας εγκατάστασης του νέου πυρήνα.

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

3. Κατεβάσαμε, έπειτα, τον πηγαίο κώδικα που μας παρέχει η debian, εκτελώντας στο guest μηχάνημα τις ακόλουθες εντολές:

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

   Η εκτέλεση αυτή κατέβασε στο directory /usr/src το αρχείο linux-source-3.2.tar.bz2

4. Ύστερα, αντιγράψαμε τον πηγαίο κώδικα του πυρήνα στο host μηχάνημα

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

   και τον κάναμε extract:

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

5. Έπειτα, κατεβάσαμε από το mycourses το αρχείο kernel_config_new.txt και το αντιγράψαμε στο untarred directory του linux-source με όνομα .config .

6. Ακολούθως, για να κάνουμε configure τον πυρήνα με το συκεκριμένο configuration αρχείο εκτελέσαμε την παρακάτω εντολή:

```
:~/esd/lab5/linux-source-3.16$ make ARCH=arm CROSS_COMPILE=/home/phoevos/x-tools/
    arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon-linux-gnueabihf-
```

   όπου, επειδή κάνουμε cross-compiling, υποδείξαμε στο make την target αρχιτεκτονική και τον cross compiler που θα χρησιμοποιήσουμε. Επιλέξαμε τις default επιλογές για όσα configurations ερωτηθήκαμε από το make.

7. Κατεβάσαμε, επιπλέον, από το mycourses το αρχείο builddeb_hf.patch με το οποίο κάναμε patch το script scripts/package/builddeb που περιλαμβάνεται στον πηγαίο κώδικα που κατεβάσαμε. Στην πραγματικότητα, κάναμε χειροκίνητα τις αλλαγές που περιλαμβάνονταν στο .patch, επειδή η εντολή patch αποτύγχανε.

8. Προκειμένου να δημιουργήσουμε τα 3 .deb πακέτα που θα ανέβουν στο qemu, εκτελέσαμε την ακόλουθη εντολή (με -j 4 για να χρησιμοποιηθούν όλοι οι διαθέσιμοι πυρήνες του υπολογιστή):

```
:~/esd/lab5/linux-source-3.16$ make ARCH=arm CROSS_COMPILE=/home/phoevos/x-tools/
    arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon-linux-gnueabihf-
        deb-pkg -j 4
```

9. Τελικά, ανεβάσαμε τα 3 αρχεία στο qemu (scp -P 22223 ...) και τα εγκαταστήσαμε με χρήση του package manager:

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

10. Για να λειτουργήσει ορθά το σύστημα με τον νέο πυρήνα, πρέπει κατά την εκκίνηση του qemu να δίνονται σαν ορίσματα τα σωστά αρχεία του image του πυρήνα και του initrd, που παράγονται στο guest μηχάνημα μετά την εγκατάσταση των .deb πακέτων. Επομένως, χρειάζεται να κατεβάσουμε τα εν λόγω αρχεία στο host μηχάνημα και να επανεκκινήσουμε τον qemu δίνοντας τα ως ορισματα στα flags kernel και initrd.

4.1 Ερωτήματα

1.
   - Πριν την εγκατάσταση του 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
     ```

   - Μετά την εγκατάσταση του νέου πυρήνα:

     ```
     root@debian-armhf:~# uname -a
     Linux debian-armhf 3.16.84 #3 SMP Wed Jan 6 21:52:21 EET 2021 armv7l GNU/Linux
     ```

2. Για την υλοποίηση του δικού μας system call ακολουθήσαμε τα εξής βήματα:

   - Πηγαίνουμε στο source του νέου μας πυρήνα και δημιουργούμε το directory hello/. Μέσα σ' αυτό δημιουργούμε το hello.c αρχείο για το system call

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

asmlinkage long sys_hello(void)
{
  int num=22;
  printk("Greeting from kernel and team no %d\n", num);
  return 0;
}
```

Listing 1: New system call to add to kernel

   καθώς και το αντίστοιχο Makefile αρχείο,

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

εξασφαλίζοντας ότι το system call μας θα μεταγλωττιστεί και θα περιέχεται στον πηγαίο κώδικα του πυρήνα μας.

- Μεταφερόμαστε στο main directory του πυρήνα μας και τροποποιούμε το Makefile αρχείο, προσθέτοντας το hello/ directory στον κανόνα core-y

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

  Έτσι ο compiler "γνωρίζει" ουσιαστικά ότι το νέο system call μας sys_hello() βρίσκεται σε αυτό το directory.

- Προσθέτουμε το νέο system call στο system call table του πυρήνα μας. Αφού ο πυρήνας τρέχει σε arm αρχιτεκτονική πηγαίνουμε στο **/arch/arm/kernel/call.S** αρχείο και προσθέτουμε την παρακάτω γραμμή

  ```
  /* 386 */ CALL(sys_hello)
  ```

  Ο κωδικός είναι 386, +1 από το τελευταίο call ενός system call πριν την νέα προσθήκη. Να σημειωθεί ότι πρέπει να προσέξουμε το system calls padding το οποίο με βάση αυτή τη γραμμή

  ```
  .equ    syscalls_padding, ((NR_syscalls + 3) & ~3) - NR_syscalls
  ```

  υπολογίζεται 388

- Πηγαίνουμε στο αρχείο **/arch/arm/include/uapi/asm/unistd.h** και προσθέτουμε την ακόλουθη γραμμή

  ```
  #define __NR_hello (__NR_SYSCALL_BASE+386)
  ```

  Για να προσθέσουμε τον αριθμό του system call μας.

- Ελέγχουμε το αρχείο **/arch/arm/include/asm/unistd.h** Για να βεβαιωθούμε ότι ο αριθμός των system calls είναι σύμφωνος με το padding.

- Προσθέτουμε το νέο system call στο system call header file. Πηγαίνουμε στο **/include/linux/syscalls.h** αρχείο και προσθέτουμε την παρακάτω γραμμή στο τέλος, πριν το #endif.

  ```
  asmlinkage long sys_hello(void);
  ```

  Έτσι ορίζεται το πρότυπο της συνάρτησης του system call μας. Ο όρος "asmlinkage" είναι ένα key word και υποδεικνύει στον compiler ότι δεν πρέπει να αναζητήσει τυχόν παραμέτρους σε καταχωρητές, αλλά στη στοίβα.

Έχοντας ολοκληρώσει τα βήματα για την προσθήκη του system call στον πυρήνα μας, τον μεταγλωττίζουμε εκ νέου ακολουθώντας τα βήματα 6-10 που αναφέρθηκαν παραπάνω.

3. Δημιουργούμε το αρχείο test.c για τη δοκιμή του system call μας.

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

#define SYS_hello 386

int main(void) {
    printf("Invoking hello system call.\n");
    long ret = syscall(SYS_hello);
    printf("Syscall returned %ld.\n", ret);
    return 0;
}
```

Listing 2: Tester file for our new system call

Το μεταγλωττίζουμε και τρέχουμε το εκτελέσιμο test.out στο qemu.

με την εντολή

```
dmesg | tail
```

επιβεβαιώνουμε ότι το system call εκτελέστηκε σωστά και έγραψε στον buffer του πυρήνα το μήνυμα.

```
root@debian-armhf:~# ./test.out
Invoking system call.
Syscall returned 0.
root@debian-armhf:~# dmesg | tail
[  104.578068] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[  110.019331] RPC: Registered named UNIX socket transport module.
[  110.029789] RPC: Registered udp transport module.
[  110.037364] RPC: Registered tcp transport module.
[  110.044485] RPC: Registered tcp NFSv4.1 backchannel transport module.
[  110.284887] FS-Cache: Loaded
[  110.830851] FS-Cache: Netfs 'nfs' registered for caching
[  111.883647] Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
[  677.640875] Greeting from kernel and team no 22
[  831.693128] Greeting from kernel and team no 22
root@debian-armhf:~#
```

Σχήμα 1: Success system call execution