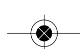


Του Γιάννη Τσιομπίκα <nuclear@member.fsf.org>

Ο Γιάννης ασχολείται ενεργά με προγραμματισμό γραφικών, system programming και kernel development.

Εισαγωγή στο kernel development

Ξεκινάμε μια σειρά άρθρων που θα σας καθοδηγήσουν στα πρώτα βήματα σε ό,τι αφορά τον προγραμματισμό πυρήνων λειτουργικών συστημάτων. Μέσα από αυτά τα άρθρα θα ξεκινήσουμε από το μηδέν να γράψουμε το δικό μας πυρήνα.

Για Smartphones

Εργαλεία: GCC, GNU make, GRUB, qemu.
Δυσκολία: ★★★★☆
URL: http://goo.gl/2yFBe

Καλώς ήρθατε στο πρώτο μέρος αυτής της σειράς άρθρων περί προγραμματισμού πυρήνων λειτουργικών συστημάτων. Ο πυρήνας ενός λειτουργικού συστήματος είναι το πρόγραμμα που διαχειρίζεται το hardware του υπολογιστή στο χαμηλότερο επίπεδο και παρέχει κάποιες βασικές υπηρεσίες στα προγράμματα των χρηστών.

Σκοπός αυτών των άρθρων δεν είναι να καλύψουμε τα πάντα γύρω από το kernel development, κάτι το οποίο είναι αδύνατο σε τόσο λίγες σελίδες, αλλά να βάλουμε τις βάσεις για τη δημιουργία ενός απλού πυρήνα, ο οποίος θα αποτελέσει εφαλτήριο για περαιτέρω πειραματισμό και απόκτηση γνώσης στον προγραμματισμό πυρήνων λειτουργικών συστημάτων.

Στη σειρά αυτή δεν θα αναφερθούμε θεωρητικά στο αντικείμενο του kernel development, αλλά θα λερώσουμε τα χέρια μας και θα αρχίσουμε να φτιάχνουμε έναν υποτυπώδη πυρήνα από το μηδέν. Ο πυρήνας αυτός, αν και δεν έχει ελπίδα να φτάσει σε σημείο να γίνει χρήσιμος και πλήρης, παρ' όλα αυτά, θα μας διδάξει κάποια βασικά πράγματα και θα μας επιτρέψει να εμβαθύνουμε στο πώς χρησιμοποιούμε και διαχειριζόμαστε το hardware του υπολογιστή.

Για τον κώδικα του πυρήνα θα χρησιμοποιήσουμε τη γλώσσα C, η οποία είναι η πλέον κατάλληλη γλώσσα για kernel development, λόγω του αρκετά low-level χαρακτήρα της. Η μοναδική ουσιαστική εναλλακτική θα ήταν να γράψουμε τον κώδικα σε assembly, αλλά αυτό θα τον έκανε πολύ πιο δυσανάγνωστο, πιο μεγάλο και αδύνατο να μεταφερθεί (όσα κομμάτια γίνεται να μεταφερθούν χωρίς να ξαναγραφτούν από το μηδέν) σε άλλη αρχιτεκτονική.

Παρ' όλα αυτά, σε κάποια σημεία του πυρήνα θα καταφύγουμε αναγκαστικά σε assembly, επειδή θα πρέπει να χρησιμοποιήσουμε συγκεκριμένες εντολές του επεξεργαστή για να επιτύχουμε κάποια πράγματα (για παράδειγμα, κατά την είσοδο και έξοδο από interrupt handlers, όπως θα δούμε στο επόμενο άρθρο).

Από τα παραπάνω προκύπτει ότι ο πυρήνας είναι εξ ορισμού άρρηκτα συνδεδεμένος με την αρχιτεκτονική του υπολογιστή στον οποίο τρέχει. Ο κώδικας που θα γράψουμε πρέπει να διαχειριστεί πολύ low-level λεπτομέρειες του hardware, μια και δεν υπάρχει άλλο abstraction layer κάτω από τον πυρήνα. Ετσι, εξαρχής πρέπει να αποφασίσουμε για ποιον υπολογιστή ακριβώς γράφουμε τον kernel.

Ο kernel που θα γράψουμε σε αυτή τη σειρά άρθρων θα τρέχει σε IBM PC - συμβατούς υπολογιστές με επεξεργαστές συμβατούς με τον Intel 386, τον οποίο θα χρησιμοποιήσουμε σε 32-bit mode.

Ο λόγος που επιλέγουμε αυτή την αρχιτεκτονική είναι καθαρά γιατί αυτό έχει ο περισσότερος κόσμος πάνω στο γραφείο του, παρά για οποιοδήποτε τεχνικό προτέρημα αυτής.

offset	μέγεθος	πεδίο
0	4	magic identifier
4	4	flags
8	4	checksum
12	4	header address
16	4	load address
20	4	load end address
24	4	bss end address
28	4	entry address
32	4	video mode type
36	4	video mode width
40	4	video mode height
44	4	video mode color depth

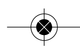
1 Το multiboot header που πρέπει να υπάρχει κάπου στην αρχή του kernel μας.

Το πρώτο άρθρο θα επικεντρωθεί στο πώς θα κάνουμε το πρόγραμμά μας να φορτωθεί και να εκτελεστεί κατά την εκκίνηση του υπολογιστή, καθώς και στη χρήση της κάρτας γραφικών για εμφάνιση κειμένου στην οθόνη. Ο κώδικας που συνοδεύει το άρθρο βρίσκεται στη σελίδα που αναφέρεται στην αρχή της πρώτης σελίδας. Προτείνεται να κατεβάσετε τον κώδικα και να τον έχετε εύκαιρο καθώς διαβάζετε το παρόν άρθρο, μια και εδώ θα συμπεριληφθούν μόνο μικρά αποσπάσματα, εξαιτίας του περιορισμένου χώρου στις σελίδες του περιοδικού.

Boot Loading

Οταν ξεκινάμε ένα PC, ή όταν κάνουμε reset, ο επεξεργαστής ξεκινάει σε 16-bit mode. Σε αυτό το mode ο επεξεργαστής δρα σαν να ήταν ένας 16bitος 8086/8088, έτσι ώστε να μπορεί να τρέξει κώδικας του 1981 γραμμένος για το IBM PC αμετάβλητος. Στο 16bit mode, λεγόμενο και «real mode», ο επεξεργαστής χρησιμοποιεί ένα πολύπλοκο σύστημα διευθυνσιοδότησης της μνήμης βασισμένο σε 12-bit segments και 16-bit offsets που συνδυάζονται για να δημιουργήσουν 20-bit διευθύνσεις μνήμης.

Πέραν του ότι αυτό το μοντέλο διευθυνσιοδότησης κάνει τον προγραμματισμό του συστήματος εφιαλτικό (όσοι γράφαμε κάποτε κώδικα στο DOS το θυμόμαστε με τρόμο), μας περιορίζει και στο μέγιστο ποσό μνήμης που μπορούμε να χρησιμοποιήσουμε, το οποίο είναι 1MB. Επιπλέον, πολύ



σημαντικό μειονέκτημα του «real mode» είναι ότι δεν μπορούμε να χρησιμοποιήσουμε τα χαρακτηριστικά του επεξεργαστή που είναι απαραίτητα για τη λειτουργία όλων των σύγχρονων λειτουργικών συστημάτων, όπως virtual memory, memory protection κ.λπ. Για να τα ενεργοποιήσουμε, πρέπει ο επεξεργαστής να μπει σε 32-bit protected mode, το οποίο πρωτοεισήγαγε στην αρχιτεκτονική της Intel ο 386.

Αλλά για να επανέλθουμε, το πρώτο πράγμα που συμβαίνει όταν ξεκινά ο επεξεργαστής τη λειτουργία του είναι να αρχίσει να εκτελεί 16-bit εντολές από την κορυφή της μνήμης. Εκεί βρίσκεται ο εναρκτήριος κώδικας του BIOS στην ROM του υπολογιστή, ο οποίος -αφού μετρήσει τη μνήμη και αρχικοποιήσει το bus με τα περιφερειακά και τους δίσκους-, φορτώνει στη μνήμη, στη διεύθυνση 0000:7c00 (segment:offset) το πρώτο sector (512 bytes) του boot δίσκου και κάνει jump εκεί.

Αυτό που κάνουν λοιπόν τα λειτουργικά συστήματα για PC είναι να τοποθετήσουν στο πρώτο sector του boot δίσκου ένα πολύ μικρό πρόγραμμα (512 bytes όπως είπαμε) του οποίου η δουλειά είναι να φορτώσει στη μνήμη τον kernel από το σημείο του δίσκου στο οποίο βρίσκεται και να κάνει jump εκεί έτσι ώστε ο επεξεργαστής να αρχίσει να εκτελεί τον κώδικα του kernel.

Κάτι τέτοιο είναι βιώσιμο όταν αναφερόμαστε σε απλούς 16-bit kernels όπως αυτοί που έτρεχαν αρχικά στα PCs (MS-DOS), όμως περιπλέκεται όταν πάμε σε πιο εξελιγμένα σύγχρονα λειτουργικά συστήματα με 32-bit kernels, οι οποίοι μάλιστα θέλουμε να έχουν την ευελιξία να είναι τοποθετημένοι «κάπου μέσα στο filesystem» και όχι απαραίτητα σε ένα προκαθορισμένο σημείο του δίσκου. Πλέον, ο boot loader είναι ένα αρκετά πολύπλοκο πρόγραμμα που πρέπει να μπορεί να βάζει τον επεξεργαστή σε 32-bit protected mode προτού δώσει τον έλεγχο στον kernel, να διαβάζει διάφορα συστήματα αρχείων, η ακόμη και να φορτώνει kernels που δεν βρίσκονται καν στον υπολογιστή αλλά κάπου στο τοπικό δίκτυο σε έναν TFTP server.

Για αυτόν το λόγο, μοντέρνοι boot loaders, όπως ο GRUB, σπάνε τη διαδικασία σε 2 (ή και περισσότερα) μέρη. Στο πρώτο sector του boot δίσκου μπαίνει ο λεγόμενος first stage boot loader, ένα πολύ απλό πρόγραμμα (για να χωρά σε 512 bytes) το οποίο φορτώνει και εκτελεί το second stage boot loader. Ο second stage boot loader που δεν έχει όρια μεγέθους, είναι ουσιαστικά ένας υποτυπώδης kernel από μόνος του, που ξέρει να διαβάζει δίσκους και filesystems και όλα όσα αναφέραμε παραπάνω, ενώ είναι σε θέση να φορτώσει τον kernel και να του περάσει τον έλεγχο, έχοντας ήδη βάλει τον επεξεργαστή σε 32-bit protected mode.

Αρχίζοντας το development

Ξεκινώντας, έχουμε δύο επιλογές: μπορούμε να γράψουμε έναν boot loader για τον kernel μας ή μπορούμε να χρησιμοποιήσουμε ένα γενικό boot loader σαν το GRUB και να αρχίσουμε κατευθείαν με τον κώδικα του kernel. Η δεύτερη επιλογή μάς παρέχει πολλά πλεονεκτήματα:

- Όλοι έχουμε ήδη εγκατεστημένο GRUB στον υπολογιστή μας αν τρέχουμε κάποιο τυπικό GNU/Linux distribution,
- Το GRUB μπορεί να φορτώσει απλά ELF binaries απευθείας από το δίσκο μας ή ακόμη και μέσω δικτύου, το οποίο απλοποιεί το development και το debugging, μια και αφενός δεν χρειάζεται να κατασκευάζουμε bootable disk images για να δοκιμάσουμε τον kernel, αφετέρου το binary μπορεί να περιέχει κανονικά debugging symbols που αναγνωρίζει ο gdb, και μπορούμε να το χρησιμοποιήσουμε με λίγο επιπλέον κόπο για remote source-leveldebugging, κάτι το οποίο πιθανόν θα δούμε σε κάποιο επόμενο άρθρο.

Ο GRUB για να μπορεί να φορτώσει διάφορους kernels ακολουθεί ένα standard που λέγεται multiboot. Αντίστοιχα, ένας kernel για να μπορεί να φορτωθεί από multiboot boot loader, όπως ο GRUB, πρέπει να ακολουθεί το ίδιο standard.

Το multiboot standard ορίζει ότι κάπου μέσα στα πρώτα 8kb του kernel image, πρέπει να τοποθετήσουμε ένα extra header που λέει στον boot loader διάφορες πληροφορίες τις οποίες χρειάζεται για να φορτώσει τον kernel και να του περάσει τον έλεγχο.

Αυτές είναι οι πληροφορίες όπως η διεύθυνση στην οποία θέλουμε να φορτωθεί κάθε section του kernel binary, η διεύθυνση στην οποία θέλουμε να μεταφερθεί η εκτέλεση μετά το φόρτωμα (entry point) κ.ά. **(βλ. σχήμα 1)**. Από αυτά τα πεδία, μόνο τα τρία πρώτα είναι απαραίτητα αν το kernel image μας είναι σε ELF format, μια και ο boot loader είναι αρκετά «έξυπνος» ώστε να βρει τις σχετικές πληροφορίες που ήδη υπάρχουν στον ELF header.

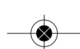
Τα τελευταία πεδία περί video modes μπορούμε να τα χρησιμοποιήσουμε για να ζητήσουμε από τον boot loader να θέσει συγκεκριμένο mode γραφικών πριν μας δώσει τον έλεγχο, κάτι το οποίο δεν θα χρησιμοποιήσουμε, αφού θέλουμε απλό text mode.

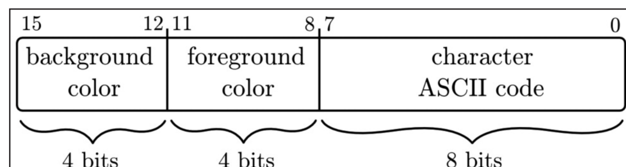
Επιπλέον το multiboot standard ορίζει και τη μέθοδο με την οποία ο boot loader μάς δίνει διάφορες κρίσιμες πληροφορίες για τον υπολογιστή, όπως το ποσό της διαθέσιμης μνήμης και ποια κομμάτια της μπορούμε να χρησιμοποιήσουμε, πληροφορίες για τους δίσκους του συστήματος κ.ά. Αυτού του είδους οι πληροφορίες θα μας χρειαστούν στα επόμενα άρθρα και θα τα δούμε λεπτομερέστερα τότε.

Προς το παρόν θα τις αγνοήσουμε.

ΚΩΔΙΚΑΣ mboot.S

```
#define MAGIC          0x1badb002
#define FLAGS          0
#define STACK_SIZE     0x4000

    .text
    .align 4
    /* multiboot header */
    .long MAGIC
    .long FLAGS
    .long -(MAGIC + FLAGS)    /* checksum */
    .fill 5, 4     /* fill out the rest with zeroes */

    .globl kentry
kentry:
    /* setup a temporary kernel stack */
    movl $(stack + STACK_SIZE), %esp
    /* reset eflags */
    pushl $0
    popf
    /* call the kernel main function */
    call kmain
    /* we dropped out of main, halt the CPU */
    cli
    hlt
```





2 Η δομή του κάθε χαρακτήρα στη video ram της VGA σε text mode.

```
    /* space for the temporary kernel stack */
    .comm stack, STACK_SIZE
```

Το αρχείο mboot.S περιέχει τον κώδικα assembly που δημιουργεί το multiboot header, καθώς και το entry point του kernel μας στο οποίο περνάει την εκτέλεση ο boot loader.

Το πρώτο πράγμα που κάνουμε μόλις πάρουμε τον έλεγχο είναι να δημιουργήσουμε ένα προσωρινό stack για τον κώδικα του kernel και να βάλουμε τον esp να δείχνει στην κορυφή του. Κατόπιν, καλούμε τη συνάρτηση kmain και εάν επιστρέψουμε ποτέ από την kmain (κάτι που όταν ολοκληρώσουμε τον πυρήνα θεωρητικά δεν θα πρέπει να συμβαίνει ποτέ) σταματάμε τον επεξεργαστή με την εντολή hlt.

Η εντολή cli, που προηγείται της hlt, απενεργοποιεί όλα τα interrupts, το οποίο κανονικά είναι απαραίτητο για να βεβαιωθούμε ότι δεν θα ξυπνήσει ξανά ο επεξεργαστής ύστερα από λίγο όταν σηκωθεί κάποιο timer interrupt ή ο χρήστης πατήσει κάποιο πλήκτρο.

Στον κώδικα, ως έχει αυτή τη στιγμή, είναι απλώς περιττή, αφού δεν ενεργοποιούμε πουθενά interrupts αλλά τη βάζουμε για να μην το ξεχάσουμε αργότερα.

Kernel main

Ο υπόλοιπος κώδικας είναι σε C. Στο αρχείο main.c έχουμε τη συνάρτηση kmain που το μόνο που κάνει προς το παρόν είναι να καλέσει την puts για να τυπώσει μερικά strings. Για να το πετύχει αυτό όμως χρειάζεται κάποια δουλειά. Σε kernel space δεν υπάρχει C library που να μας παρέχει I/O και άλλα βοηθήματα. Ο,τι χρειαζόμαστε από τη libc πρέπει να το γράψουμε μόνοι μας. Δεν υπάρχει stdio.h με τη γνώριμη printf, ούτε string.h με τις γνωστές συναρτήσεις για επεξεργασία αλφαριθμητικών. Αλλά κανείς δεν μας εμποδίζει να τα γράψουμε εφόσον τα χρειαζόμαστε, και αυτό θα κάνουμε. Μέσα στον κατάλογο src/klibc αρχίζουμε να προσθέτουμε τις συναρτήσεις της βιβλιοθήκης C που νομίζουμε ότι μας είναι χρήσιμες.

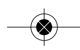
Στη συγκεκριμένη περίπτωση, έχουμε γράψει τις memset, memcpy, memmove στο αρχείο klibc/string.c και τη συνάρτηση puts στο klibc/stdio.c. Η puts καλεί απλά την putchar για να τυπώσει έναν-έναν όλους τους χαρακτήρες την οποία την έχουμε υλοποιήσει στο αρχείο term.c. Η putchar, με τη σειρά της, αφού χειριστεί ειδικούς χαρακτήρες όπως '\n' ή '\t' επηρεάζοντας την ενεργή θέση του cursor, καλεί πιο low-level συναρτήσεις όπως η set_char, scroll_scr και set_cursor που είναι υλοποιημένες στο αρχείο vid.c και ξέρουν πώς να χειριστούν το hardware της VGA για να κάνουν τη δουλειά τους.

VGA driver

Πώς όμως τελικά χειριζόμαστε τη VGA για να γράψουμε κείμενο στην οθόνη; Ολος ο σχετικός κώδικας βρίσκεται στο αρχείο vid.c:

ΚΩΔΙΚΑΣ vid.c

```
#include <string.h>
#include "vid.h"
#include "asmops.h"

#define WIDTH 80
#define HEIGHT 25

/* CRTC ports */
#define CRTC_ADDR   0x3d4
#define CRTC_DATA   0x3d5

/* CRTC registers */
#define CRTC_CURSOR_HIGH 0xe
#define CRTC_CURSOR_LOW  0xf

/* construct a character with its attributes */
#define VMEM_CHAR(c, fg, bg) \
    ((uint16_t)(c) | (((uint16_t)(fg) & 0xf) << 8) | \
     (((uint16_t)(bg) & 0xf) << 12))

#define CLEAR_CHAR      VMEM_CHAR(' ', LTGRAY, BLACK)

/* pointer to the text mode video memory */
static uint16_t *vmem = (uint16_t*)0xb8000;

void clear_scr(void)
{
    memset16(vmem, CLEAR_CHAR, WIDTH * HEIGHT);
}

void set_char(char c, int x, int y, int fg, int bg)
{
    vmem[y * WIDTH + x] = VMEM_CHAR(c, fg, bg);
}

void set_cursor(int x, int y)
{
        int loc;
    if(x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT) {
        loc = 0xffff;
    } else {
        loc = y * WIDTH + x;
    }

    /* tell the vga where we want the cursor by writing
     * to the "cursor address" register of the CRTC */
    outb(CRTC_CURSOR_LOW, CRTC_ADDR);
    outb(loc, CRTC_DATA);
    outb(CRTC_CURSOR_HIGH, CRTC_ADDR);
    outb(loc >> 8, CRTC_DATA);
}

void scroll_scr(void)
{
    /* simple scrolling by manually copying memory */
    memmove(vmem, vmem + WIDTH, WIDTH * (HEIGHT - 1)
 * 2);
    memset16(vmem + WIDTH * (HEIGHT - 1), CLEAR_CHAR,
 WIDTH);
}
```

Ο συγκεκριμένος κώδικας είναι πολύ απλοϊκός αλλά δουλεύει για τις ανάγκες μας. Κατ' αρχάς, η μνήμη από τη διεύθυνση b8000 και για τα επόμενα 4.000 bytes ανήκει στη VGA και περιέχει τους χαρακτήρες που εμφανίζονται στην οθόνη. Για κάθε χαρακτήρα χρησιμοποιούμε 2 bytes. Το χαμηλότερο byte είναι ο κωδικός του χαρακτήρα στο ASCII και το υψηλότερο byte ορίζει το χρώμα του χαρακτήρα και το χρώμα του υποβάθρου **(βλ. σχήμα 2).**

Το macro VMEM_CHAR κατασκευάζει ένα 2-byte integer που περιέχει το χαρακτήρα με τα attributes που θέλουμε έτοιμο να τοποθετηθεί στη video memory. H set_char υπολογίζει σε ποια διεύθυνση ακριβώς πρέπει να τοποθετηθεί ο χαρακτήρας και τον βάζει εκεί, χρησιμοποιώντας την προαναφερθείσα VMEM_CHAR.

H scroll_scr υλοποιεί scrolling προς τα πάνω κατά μία γραμμή απλά αντιγράφοντας όλα τα περιεχόμενα της μνήμης πλην της πρώτης γραμμής, μία γραμμή πιο πάνω, και κατόπιν καθαρίζει την τελευταία γραμμή. H συνάρτηση memset16 είναι στα πρότυπα της standard memset, αλλά γεμίζει το κομμάτι μνήμης που θέλουμε με 16-bit integers αντί για 8-bit chars.

Το πιο δυσνόητο κομμάτι σε αυτό τον κώδικα που χρήζει περαιτέρω επεξήγησης, είναι ίσως η συνάρτηση set_cursor. Αυτό που θέλουμε να πετύχουμε είναι να μπορούμε να ορίσουμε σε ποιο σημείο της οθόνης εμφανίζεται ο cursor, έτσι ώστε να τον θέτουμε πάντα στο σημείο που θα εισαχθεί ο επόμενος χαρακτήρας, για να έχει ένα οπτικό βοήθημα ο χρήστης όταν πληκτρολογεί.

H VGA έχει μια σειρά από registers που μας δίνουν χρήσιμες πληροφορίες, ενώ παράλληλα μας αφήνουν να επηρεάσουμε τη λειτουργία της. Στην προκειμένη περίπτωση, η λειτουργία που μας ενδιαφέρει είναι σπασμένη σε 2 8-bit registers του CRTC (Cathode Ray Tube Controller), τον «cursor location low register» και τον «cursor location high register» με αριθμούς 0ch και 0dh αντίστοιχα. Για να γράψουμε σε κάποιον από τους registers του CRTC αρκεί να κάνουμε out στο port 3d4 τον αριθμό του register που μας ενδιαφέρει, ακολουθούμενο από ένα out στο port 3d5 με την τιμή που θέλουμε να θέσουμε.

Τα outb που καλούμε στον κώδικα παραπάνω για αυτόν το σκοπό, είναι macros με in-line assembly δηλωμένα στο asmops.h που μας επιτρέπουν να χρησιμοποιήσουμε εύκολα τις in/out εντολές του επεξεργαστή από C κώδικα.

```
#define outb(src, port) asm volatile( \
    "outb %0, %1\n\t" \
    :: "a" ((unsigned char)(src)), \
      "dN" ((unsigned short)(port)))
```

Δοκιμάζοντας τον πυρήνα

Για να κάνουμε compile τα source files του πυρήνα μας αρκεί να καλέσουμε τον gcc με παραμέτρους:

```
-m32 -nostdinc -fno-builtin -Isrc -Isrc/klibc.
```

Εν ολίγοις, λέμε στον compiler να παραγάγει 32-bit κώδικα, ακόμη και αν είναι 64-bit compiler, να μην ψάξει στο /usr/include για headers, παρά μόνο στα directories src και src/klibc που έχουμε τα δικά μας headers, και να μη χρησιμοποιήσει δικές του εκδόσεις κάποιων standard C συναρτήσεων όπως συνηθίζει για βελτιστοποίηση του κώδικα, αλλά να καλέσει τις δικές μας.

Στο linker πρέπει να περάσουμε το πού θέλουμε να θεωρεί ότι θα βρίσκεται το text section (ο κώδικας) του kernel στη μνήμη, κάτι το οποίο υπακούει ο GRUB και όντως το φορτώνει εκεί, μια και αυτή η τιμή μπαίνει στον ELF header. Διαλέγουμε τη διεύθυνση που αντιστοιχεί στο 1MB (-Ttext 0x100000) για να είμαστε σίγουρα πάνω από το σημείο της μνήμης που καταλαμβάνει η VGA και το BIOS.

Επίσης, πρέπει να ορίσουμε και το entry point του kernel (-e kentry). Το makefile που έρχεται με το συνοδευτικό κώδικα περιλαμβάνει μαζεύει όλα τα source files του kernel μας και τα κάνει compile και link με τις σωστές παραμέτρους.

Για να τρέξουμε τον kernel μας -και φυσικά να δούμε τι κάναμε- μπορούμε να φτιάξουμε ένα καινούργιο entry στο αρχείο menu.lst του GRUB κάπως έτσι:

```
title my kernel
root (hd0,0)
kernel /home/myuser/mykernel/kernel.elf
```

Όσο ωραίο και αν είναι να τρέχεις τον kernel που μόλις έγραψες στο πραγματικό μηχάνημα, πρακτικά είναι πολύ πιο βολικό όσο γράφουμε κώδικα να δοκιμάζουμε τον kernel σε virtual machine. Συγκεκριμένα, το qemu είναι πολύ βολικό γιατί δεν χρειάζεται καν να φτιάξουμε εικονικό bootable δίσκο με GRUB και τον kernel μας. Το qemu μπορεί να παίξει το ρόλο του multiboot-compliant boot loader και απλά να τρέξει τον kernel μας περνώντας το όνομα του ELF εκτελέσιμου σαν παράμετρο στο qemu ως εξής:

```
qemu -kernel kernel.elf
```

Παρ' όλα αυτά, καλό είναι να δοκιμάζουμε πού και πού και στο πραγματικό μηχάνημα γιατί, εάν δεν είμαστε προσεκτικοί, μπορεί ο πυρήνας μας να καταλήξει να δουλεύει μόνο στο virtual machine.

•

Ασκήσεις για τον αναγνώστη

Μέχρι το επόμενο τεύχος, μπορείτε να δοκιμάσετε το χέρι σας σε δύο ενδιαφέρουσες προσθήκες στον κώδικα:

- Υλοποίηση μίας printf συνάρτησης για πιο εύχρηστο output και
- Καλύτερη υλοποίηση της scroll_scr, χρησιμοποιώντας τις δυνατότητες της VGA για hardware scrolling, αντί να αντιγράφουμε αράδες χειροκίνητα.

Για την πρώτη προσθήκη, θα χρειαστεί να βρείτε τρόπο να υλοποιήσετε τα va_start, va_arg κ.λπ. που παρέχει κανονικά η βιβλιοθήκη C στο stdarg.h, ενώ για τη δεύτερη χρειάζεται να ανατρέξετε στην περιγραφή των CRTC registers της VGA.

Στείλτε τις απαντήσεις σας στο nuclear@member.fsf.org.

Οι καλύτερες προσθήκες ή διορθώσεις αναγνωστών θα δημοσιευθούν στο επόμενο άρθρο και σε κάθε περίπτωση κώδικας που υλοποιεί τις παραπάνω προτάσεις θα υπάρχει στο συνοδευτικό κώδικα του επόμενου άρθρου.

Περαιτέρω αναγνώσματα

- «Operating Systems: Design and Implementation», Andrew Tanenbaum.
  Εκδόσεις: Prentice Hall.
- Multiboot specification
  http://www.gnu.org/software/grub/manual/multiboot/multiboot.html
- FreeVGA Project: VGA Chipset Reference
  http://www.osdever.net/FreeVGA/vga/vga.htm