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

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

Εισαγωγή στο kernel development

Σε αυτό το τεύχος θα κλείσουμε το θέμα του memory management, υλοποιώντας μία malloc για τον πυρήνα, και θα ασχοληθούμε με το χρόνο, τους timers και το real time clock.

Για Smartphones

Εργαλεία: GCC, GNU make, GRUB, qemu

Δυσκολία: ★★★★☆

URL: http://nuclear.sdf-eu.org/articles/kerneldev/

Καλώς ήρθατε σε άλλο ένα άρθρο της σειράς προγραμματισμού πυρήνων λειτουργικών συστημάτων από το μηδέν. Στα προηγούμενα επεισόδια ξεκινήσαμε από έναν άδειο editor buffer και «επιτεθήκαμε», ένα προς ένα, σε πολλά από τα βασικά στοιχεία που πρέπει να φτιάξουμε σε έναν πυρήνα. Είδαμε τη διαδικασία εκκίνησης του υπολογιστή και πώς μπορούμε να τον πείσουμε να φορτώσει το πρόγραμμά μας στη μνήμη και να μας περάσει τον έλεγχο. Γράψαμε έναν απλό VGA driver, για να μπορούμε να γράψουμε κείμενο στην οθόνη με τη δική μας printf. «Βουτήξαμε» στην παράξενη αρχιτεκτονική της Intel και είδαμε ποια είναι τα απαραίτητα segments που πρέπει να χρησιμοποιήσουμε και πώς να τα φέρουμε στα μέτρα μας, δημιουργώντας ένα ενιαίο γραμμικό διάστημα διευθύνσεων όπως το θέλουμε. Μάθαμε πώς χειριζόμαστε interrupts από τον ίδιο τον επεξεργαστή, αλλά και από τις συσκευές του συστήματος. Είδαμε πώς δουλεύει το paging hardware του επεξεργαστή για τη μετάφραση εικονικών διευθύνσεων σε φυσικές διευθύνσεις και πώς κάνουμε map και unmap pages με τη βοήθεια του page table. Τέλος, υλοποιήσαμε διαχειριστές μνήμης που δεσμεύουν και αποδεσμεύουν physical και virtual pages για τις ανάγκες του συστήματός μας.

> **Στα επόμενα τεύχη έχουμε και πάλι να «επιτεθούμε» σε κάποια ογκώδη και σημαντικά κομμάτια, όπως διεργασίες, multitasking και συστήματα αρχείων.**

Όλα τα παραπάνω ήταν σημαντικές βάσεις για τη συνέχεια αυτού του project και σε κάθε τεύχος ανεβάζαμε όλο και περισσότερο την πολυπλοκότητα των ιδεών που συζητούσαμε από αυτές τις σελίδες. Στα επόμενα τεύχη έχουμε και πάλι να «επιτεθούμε» σε κάποια ογκώδη και σημαντικά κομμάτια, όπως διεργασίες, multitasking και συστήματα αρχείων. Όμως, εν μέσω καλοκαιριού επιβάλλεται να πάρουμε μία ανάσα, να κατεβάσουμε ταχύτητες, να χάσουμε κανένα αστέρι από την επικεφαλίδα και να προετοιμάσουμε το έδαφος για ό,τι έπεται.

Σε αυτό το άρθρο, κατ' αρχάς, θα κλείσουμε το κεφάλαιο της διαχείρισης μνήμης, περιγράφοντας την υλοποίηση των malloc και free στην klibc, που πατάνε πάνω στον page allocator του προηγούμενου τεύχους, για να μας δώσουν fine–grained διαχείριση της μνήμης του πυρήνα. Μετά θα δούμε πώς χρησιμοποιούμε τον timer του συστήματος για να λαμβάνουμε interrupts σε τακτά διαστήματα, και να κρατάμε το χρόνο με υψηλή ανάλυση. Και, τέλος, θα δούμε πώς διαχειριζόμαστε την ημερομηνία και την ώρα και πώς διαβάζουμε από το real time clock του υπολογιστή.

Αυτή τη φορά δεν προλάβαμε να βάλουμε τον κώδικα του άρθρου στο συνοδευτικό DVD του περιοδικού, αλλά μπορείτε, όπως πάντα, να τον κατεβάσετε από το Web site της σειράς, http://nuclear.sdf-eu.org/articles/kerneldev. Στο ίδιο site θα βρείτε και τα προηγούμενα άρθρα της σειράς, διαθέσιμα υπό τους όρους της άδειας Creative Commons Attribution Share-Alike.

Kernel malloc

Στο προηγούμενο άρθρο υλοποιήσαμε τη συνάρτηση pgalloc, που βρίσκει ένα ελεύθερο συνεχές κομμάτι απ' όσα virtual pages ζητηθούν, τα οποία μας παρέχει αφού πρώτα τα κάνει map σε αντίστοιχο αριθμό από physical pages. Για τη δέσμευση των virtual pages είδαμε ότι μπορούμε να κρατάμε μία λίστα από διαστήματα συνεχών ελεύθερων pages, την οποία διασχίζουμε ψάχνοντας το πρώτο διάστημα με αρκετά pages ώστε να καλύψει τις ανάγκες μας (first fit). Την ίδια ακριβώς στρατηγική μπορούμε να ακολουθήσουμε και για τη malloc (klibc/malloc.c), με μοναδική διαφορά ότι αντί να χειριζόμαστε διαστήματα από pages, χειριζόμαστε διαστήματα από bytes σε pages που έχουμε ήδη δεσμεύσει, καλώντας την pgalloc.

Άλλη μία διαφορά προκύπτει από το ότι δεν θέλουμε να χρειάζεται να περάσουμε ως παράμετρο της free το μέγεθος της μνήμης που ελευθερώνουμε, οπότε ο allocator πρέπει να γνωρίζει με κάποιον τρόπο τι μέγεθος είχε κάθε κομμάτι μνήμης που μοίρασε. Η απλούστερη μέθοδος για να έχουμε διαθέσιμη αυτή την πληροφορία, είναι να δεσμεύουμε κάθε φορά λίγο μεγαλύτερο κομμάτι από το ζητούμενο, έτσι ώστε να γράφουμε αμέσως πριν από την αρχή του σε γνωστό αρνητικό offset το μέγεθος που δεσμεύθηκε. Το struct alloc_desc παίζει το ρόλο αυτού του header και περιέχει, εκτός από το μέγεθος του allocation, και μία γνωστή τιμή η οποία ελέγχεται από τη free. Αν δεν βρεθεί αυτή η τιμή, τότε σίγουρα κάποιο σοβαρό προγραμματιστικό λάθος έχει συμβεί: Είτε προσπαθήσαμε να ελευθερώσουμε από κάποιο pointer που δεν πήραμε από τη malloc ή κατά λάθος γράψαμε πάνω από τον allocator descriptor. Σε αυτή την περίπτωση, ο kernel πανικοβάλλεται, ώστε να μην περάσει απαρατήρητο το bug.

```
#define MAGIC    0xbaadbeef

struct alloc_desc {
    size_t size;
    uint32_t magic;
};
```

Η free απλώς προσθέτει το range που ελευθερώνουμε στη free_list και καλεί την coalesce που ελέγχει αν αυτό το διάστημα γειτονεύει με κάποιο άλλο ελεύθερο διάστημα, ώστε να μεγαλώσει εκείνο αντί να προσθέσει άλλο ένα node. Αυτό είναι σημαντικό, γιατί αλλιώς θα κατέληγε η free_list να περιέχει έναν μεγάλο αριθμό μικρών γειτονικών διαστημάτων, τα

LINUX LABS KERNEL 90-93_PRODUCTIVITY 17/06/2011 11:09 ΠΜ Page 91

οποία πρακτικά δεν θα μπορούσαν να χρησιμοποιηθούν ξανά, παρά μόνο για πολύ μικρά allocations. Το ίδιο, φυσικά, είχαμε κάνει και στο προηγούμενο άρθρο για την pgfree.

Σε αυτή τη φάση, η υλοποίηση της free δεν αποδεσμεύει ποτέ pages, προτιμώντας να τα κρατήσει στη free_list για μελλοντικά allocations. Αν δούμε στο μέλλον ότι αυτό προκαλεί προβλήματα εξάντλησης μνήμης, μπορούμε να αλλάξουμε αυτή τη στρατηγική και να ελευθερώνουμε, παραδείγματος χάρη, αν ξεμένουν στη free_list, πάνω από τα 2/3 των pages που δεσμεύσαμε.

Timers

Μία πολύ σημαντική λειτουργία ενός πυρήνα είναι το timekeeping, δηλαδή, η μέτρηση του χρόνου. Κατ' αρχάς, η χρονομέτρηση χρησιμοποιείται από πολλά user programs και πρέπει ο πυρήνας να παρέχει με κάποιον τρόπο τις σχετικές υπηρεσίες. Αλλά ακόμη πιο σημαντική είναι η χρήση των timers σε ένα time sharing σύστημα, γιατί πρέπει να μπορεί ο πυρήνας να διακόψει την εκτέλεση του ενεργού process ανά τακτά διαστήματα, για να δώσει χρόνο σε κάποιο άλλο, έτσι ώστε να δημιουργείται η ψευδαίσθηση ότι εκτελούνται ταυτόχρονα. Όλοι οι υπολογιστές διαθέτουν κάποιον hardware timer που σηκώνει ένα interrupt με συγκεκριμένο ρυθμό, το οποίο μπορούμε να χρησιμοποιήσουμε για να κρατάμε χρόνο στον πυρήνα. Το IBM PC είχε γι' αυτή τη δουλειά ένα chip της Intel, τον 8253 Programmable Interval Timer (PIT), ο οποίος πλέον υλοποιείται ως μέρος του chipset της motherboard και όχι ως ξεχωριστό εξάρτημα.

Το Intel 8253 παίρνει ως είσοδο ένα clock signal που προέρχεται από κάποιον κρύσταλλο και σε κάθε παλμό μειώνει έναν 16bit binary counter, του οποίου την αρχική τιμή μπορούμε να θέσουμε προγραμματιστικά. Όταν ο counter φτάσει στο μηδέν, ενεργοποιείται το output, το οποίο στο PC είναι συνδεδεμένο με το IRQ 0 pin του 8259 PIC (Programmable Interrupt Controller), ο οποίος με τη σειρά του σηκώνει interrupt στον επεξεργαστή και διακόπτεται η κανονική εκτέλεση, για να πάρει τον έλεγχο ο interrupt handler. Στο mode 3 (square wave generator) όπου θα χρησιμοποιήσουμε τον timer, όταν μηδενιστεί ο counter, ξεκινά από την αρχική τιμή που είχαμε ορίσει. Υπάρχουν και άλλα modes λειτουργίας, όπως τα "one-shot", και "strobe", τα οποία, όμως, δεν μας είναι εξίσου χρήσιμα για τις ανάγκες του πυρήνα και δεν θα αναφερθούμε σε αυτά.

> **Όλοι οι υπολογιστές διαθέτουν κάποιον hardware timer που σηκώνει ένα interrupt με συγκεκριμένο ρυθμό, το οποίο μπορούμε να χρησιμοποιήσουμε για να κρατάμε χρόνο στον πυρήνα.**

Ακριβέστερα, το 8253 έχει όχι έναν, αλλά τρεις counters, από τους οποίους μόνο ο counter 0 είναι συνδεδεμένος με κάποιο IRQ line και μπορεί να χρησιμοποιηθεί για να κρατάμε χρόνο στον πυρήνα. Ο counter 1 ιστορικά οδηγούσε το refresh της DRAM και δεν μπορεί να χρησιμοποιηθεί, ενώ ο counter 2 είναι συνδεδεμένος με το PC speaker, ώστε να παράγει ηχητικούς τόνους σε όποια συχνότητα θέλουμε.

Ουσιαστικά, θέτοντας την αρχική τιμή του counter, πετυχαίνουμε διαίρεση της συχνότητας του αρχικού clock signal του κρυστάλλου, ο οποίος πάλλεται στα 1.193182MHz. Για παράδειγμα, αν θέσουμε στον counter αρχική τιμή 2, τότε θα χρειαστούν δύο παλμοί του clock για να φτάσει στο μηδέν και να σηκωθεί το interrupt, άρα τα interrupts συμβαίνουν με τη μισή συχνότητα (596.591KHz). Η μέγιστη τιμή που μπορούμε να θέσουμε στον counter, όντας 16bit, είναι 65535, το οποίο μας δίνει την ελάχιστη δυνατή συχνότητα παραγωγής interrupts: 1193182 / 65535 = 18.20679Hz περίπου.

Η συνάρτηση init_timer (timer.c) καλείται για να αρχικοποιήσει τον counter 0 του PIT, βάζοντάς τον στο mode λειτουργίας που θέλουμε και θέτοντας την αρχική τιμή που χρειάζεται, ώστε να πετύχουμε το επιθυμητό frequency.

```
/* frequency of the timer ticks in hertz */
#define TICK_FREQ_HZ    250
/* frequency of the input clock signal */
#define OSC_FREQ_HZ     1193182
/* macro to divide and round to the nearest int */
#define DIV_ROUND(a, b) \
    ((a) / (b) + ((a) % (b)) / ((b) / 2))
#define PORT_DATA0          0x40
#define PORT_CMD            0x43
#define CMD_ACCESS_BOTH   (3 << 4)
#define CMD_OP_SQWAVE     (3 << 1)

void init_timer(void)
{
    /* calculate the reload count: round(osc / freq) */
    int reload = DIV_ROUND(OSC_FREQ_HZ, TICK_FREQ_HZ);

    /* set the mode to square wave for counter 0
     * low & high reload count bytes follow */
    outb(CMD_ACCESS_BOTH | CMD_OP_SQWAVE,
PORT_CMD);
    outb(reload & 0xff, PORT_DATA0);
    outb((reload >> 8) & 0xff, PORT_DATA0);

    interrupt(IRQ_TO_INTR(0), intr_handler);
}

static void intr_handler()
{
    nticks++;
}
```

Η init_timer, κατ' αρχάς, υπολογίζει την αρχική τιμή που πρέπει να θέσει, διαιρώντας τη συχνότητα του clock με την επιθυμητή συχνότητα των timer ticks. Το πρώτο outb θέτει τον counter 0 σε square wave mode και ορίζει ότι θα ακολουθήσουν δύο bytes στο data port με την αρχική τιμή για τον counter, τα οποία στέλνονται αμέσως μετά. Τέλος, η κλήση στην interrupt (intr.c) ορίζει τη συνάρτηση intr_handler ως χειριστή των interrupts που προέρχονται από IRQ 0. Η intr_handler απλώς αυξάνει κάθε φορά τη μεταβλητή nticks κατά ένα, ώστε να ξέρουμε πόσα ticks (1/250 του δευτερολέπτου) πέρασαν από την εκκίνηση του kernel.

Πραγματικός χρόνος

Μια και μιλήσαμε για τη μέτρηση του χρόνου, είναι αδύνατο να μην αναφερθούμε και σε άλλη μία πολύ βασική λειτουργία, που είναι η τήρηση του πραγματικού χρόνου, δηλαδή, της ημερομηνίας και ώρας. Είναι απαραίτητο ο πυρήνας να γνωρίζει ανά πάσα στιγμή τον πραγματικό χρόνο, ώστε να μπορεί να τον παρέχει στα προγράμματα.

Υπάρχουν πολλοί τρόποι για να κρατά ο πυρήνας τον πραγματικό χρόνο και κάθε σύστημα το κάνει διαφορετικά.

Το τσιπάκι στατικής μνήμης για το RTC του IBM PC.

Εμείς θα ακολουθήσουμε τον τρόπο του UNIX, το οποίο κρατάει το χρόνο σε δευτερόλεπτα από το UNIX epoch, δηλαδή, από την 1/1/1970. Κατά το initialization θα διαβάσουμε την ημερομηνία και την ώρα από το ρολόι του υπολογιστή και θα τα μετατρέψουμε σε δευτερόλεπτα από το epoch. Έτσι, όταν χρειαστεί να υπολογίσουμε το χρόνο, αρκεί να προσθέσουμε τα δευτερόλεπτα που πέρασαν από την εκκίνηση (nticks/TICK_FREQ_HZ), στον αρχικό χρόνο.

Όλοι οι υπολογιστές έχουν κάποιου είδους ρολόι πραγματικού χρόνου (RTC) που δουλεύει με μπαταρία ακόμη και αν ο υπολογιστής είναι σβηστός και αποσυνδεμένος από την παροχή ρεύματος. Το RTC στον IBM PC κρατάει την ημερομηνία και την ώρα σε ένα chip στατικής μνήμης συνεχώς τροφοδοτούμενο από την μπαταρία, το οποίο ιστορικά λέγεται CMOS RAM.

> **Όλοι οι υπολογιστές έχουν κάποιου είδους ρολόι πραγματικού χρόνου (RTC) που δουλεύει με μπαταρία ακόμη και αν ο υπολογιστής είναι σβηστός και αποσυνδεμένος από την παροχή ρεύματος.**

Η ημερομηνία και η ώρα είναι χωρισμένες σε μία σειρά από registers που διαβάζουμε χρησιμοποιώντας τα κλασικά I/O instructions του x86. Πρώτα γράφουμε στο control port (0x70) το νούμερο του register που μας ενδιαφέρει και μετά διαβάζουμε από το data port (0x71) την τιμή του register που διαλέξαμε. Η συνάρτηση read_reg (rtc.c) κάνει αυτή τη δουλειά.

```
static int read_reg(int reg)
{
    unsigned char val;
    outb(reg, PORT_CTL);
    iodelay();
    inb(val, PORT_DATA);
    iodelay();
    return val;
}
```

Τα iodelay (asmops.h) υπάρχουν απλώς για να δώσουν αρκετό χρόνο. στο CMOS να πραγματοποιήσει την αλλαγή register προτού διαβάσουμε την τιμή. Τα νούμερα των registers που περνάμε στη read_reg είναι δηλωμένα στο rtc.c και, για παράδειγμα, ο register 9 περιέχει το έτος, ο register 8 το μήνα κ.λπ.

Φυσικά, τίποτε στην ταλαιπωρημένη αρχιτεκτονική του PC δεν είναι τόσο απλό. Κατ' αρχάς, η ώρα μπορεί να είναι είτε σε 24ωρη μορφή ή σε 12ωρη, με το πιο σημαντικό bit να καθορίζει αν πρόκειται για προ μεσημβρίας ή μετά μεσημβρίαν.

Επίσης, ο χρόνος μπορεί να είναι είτε σε πλήρη μορφή (2011) ή σε δύο ψηφία μόνο (11), με πιο σύνηθες το δεύτερο. Τέλος, οι τιμές που διαβάζουμε απ' όλους τους RTC registers μπορεί να είναι είτε κανονικοί δυαδικοί ακέραιοι ή σε μορφή BCD (binary coded decimal), όπου κάθε nibble (4bit κομμάτι) χρησιμοποιείται σαν δεκαδικό ψηφίο.

Τη μορφή της ώρας και το αν οι τιμές είναι BCD ή binary, τα μαθαίνουμε από τα bits 1 και 2 του status B register. Η συνάρτηση read_rtc (rtc.c) χειρίζεται όλες τις παραπάνω περιπτώσεις, διαβάζει όλους τους σχετικούς RTC CMOS registers και γεμίζει ένα standard "struct tm", το οποίο δηλώσαμε στο klibc/time.h.

```
#define BCD_TO_BIN(x) \
        ((((x) >> 4) & 0xf) * 10 + ((x) & 0xf))

static void read_rtc(struct tm *tm)
{
    int statb, pm;
    /* wait for any clock updates to finish */
    while(read_reg(REG_STATA) & STATA_BUSY);

    tm->tm_sec = read_reg(REG_SEC);
    tm->tm_min = read_reg(REG_MIN);
    tm->tm_hour = read_reg(REG_HOUR);
    tm->tm_mday = read_reg(REG_DAY);
    tm->tm_mon = read_reg(REG_MONTH);
    tm->tm_year = read_reg(REG_YEAR);

    /* in 12hour mode, bit 7 means post-meridiem */
    pm = tm->tm_hour & HOUR_PM_BIT;
    tm->tm_hour &= ~HOUR_PM_BIT;

    /* convert to binary if needed */
    statb = read_reg(REG_STATB);
    if(!(statb & STATB_BIN)) {
        tm->tm_sec = BCD_TO_BIN(tm->tm_sec);
        tm->tm_min = BCD_TO_BIN(tm->tm_min);
        tm->tm_hour = BCD_TO_BIN(tm->tm_hour);
        tm->tm_mday = BCD_TO_BIN(tm->tm_mday);
        tm->tm_mon = BCD_TO_BIN(tm->tm_mon);
        tm->tm_year = BCD_TO_BIN(tm->tm_year);
    }
    /* make the year an offset from 1900 */
    if(tm->tm_year < 100) {
        tm->tm_year += 100;
    } else {
        tm->tm_year -= 1900;
    }
    /* if tm_hour is in 12h mode, convert to 24h */
    if(!(statb & STATB_24HR)) {
        if(tm->tm_hour == 12) {
            tm->tm_hour = 0;
        }
        if(pm) {
            tm->tm_hour += 12;
        }
    }
    tm->tm_mon -= 1;  /* make months start from 0 */
```

LINUX LABS KERNEL 90-93_PRODUCTIVITY 17/06/2011 11:10 ΠΜ Page 93


```
}
```

Τέλος, η init_rtc (rtc.c) που καλείται από τη main του πυρήνα, καλεί κατ' αρχάς τη read_rtc που μόλις είδαμε και μετά δίνει το tm structure που γέμισε η read_rtc στη mktime, για να το μετατρέψει σε δευτερόλεπτα από το epoch, τα οποία κρατά στην global μεταβλητή start_time που είναι προσβάσιμη απ' όλο τον υπόλοιπο kernel.

Standard C συναρτήσεις χρόνου

Φυσικά, οι συναρτήσεις mktime και gmtime που μετατρέπουν από struct tm σε δευτερόλεπτα από το epoch και το αντίστροφο, όπως και όλες οι συναρτήσεις της standard C library, δεν υπάρχουν αν δεν τις φτιάξουμε μόνοι μας στην klibc.

Δυστυχώς το προφανές, που θα ήταν να πολλαπλασιάσουμε τα χρόνια με τον αριθμό δευτερολέπτων ανά χρόνο, μετά τους μήνες με τον αριθμό δευτερολέπτων ανά μήνα κ.λπ., δεν λειτουργεί, γιατί πρέπει να συνυπολογίσουμε τα δίσεκτα έτη (leap years), στα οποία ο Φεβρουάριος έχει 29 αντί για 28 ημέρες.

Δίσεκτα κατά το Γρηγοριανό ημερολόγιο είναι όλα τα χρόνια που διαιρούνται ακριβώς με το 4 ή με το 400, εξαιρουμένων αυτών που διαιρούνται ακριβώς με το 100. Η συνάρτηση is_leap_year στο klibc/time.c μάς απαντά στο εάν κάποιο έτος είναι δίσεκτο ή όχι.

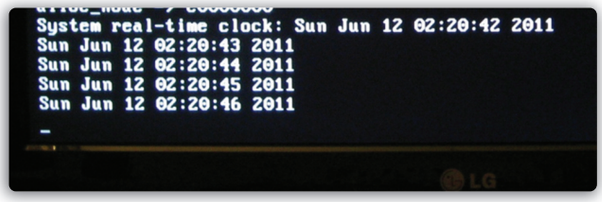
Η day_of_year στο ίδιο αρχείο,παίρνει ως παράμετρο το έτος, το μήνα και την ημέρα του μήνα και μας επιστρέφει ποια ημέρα του χρόνου είναι αυτή, λαμβάνοντας υπόψη αν είναι δίσεκτο ή όχι το έτος για το οποίο μιλάμε.

Χρησιμοποιώντας τα παραπάνω, η mktime μπορεί να υλοποιηθεί ως εξής:

```
#define MINSEC      60
#define HOURSEC     (60 * MINSEC)
#define DAYSEC      (24 * HOURSEC)
#define YEARDAYS(x) (is_leap_year(x) ? 366 : 365)
#define EPOCH_WDAY  4  /* 1-1-1970 was thursday */

time_t mktime(struct tm *tm)
{
    int i, num_years = tm->tm_year - 70;
    int year = 1970;
    int days = day_of_year(tm->tm_year + 1900,
            tm->tm_mon, tm->tm_mday - 1);
    tm->tm_yday = days;  /* fix yearday */

    for(i=0; i<num_years; i++) {
        days += YEARDAYS(year++);
    }
    /* fix weekday */
    tm->tm_wday = (days + EPOCH_WDAY) % 7;

    return days * DAYSEC + tm->tm_hour * HOURSEC +
        tm->tm_min * MINSEC + tm->tm_sec;
}
```

Η gmtime κάνει το αντίστροφο, δηλαδή, παίρνει δευτερόλεπτα από το epoch και μας γεμίζει ένα struct tm. Λόγω χώρου, δεν θα τη συμπεριλάβουμε εδώ, δείτε τον κώδικα στο klibc/time.c. Συνοπτικά, αυτό που κάνει είναι να ξεκινά υπολογίζοντας πόσες μέρες αντιστοιχούν στα δευτερόλεπτα που της δίνουμε (t / DAYSEC). Μετά, ξεκινώντας από το 1970, μειώνει από αυτές τις μέρες τον αριθμό των ημερών που αντιστοιχούν σε κάθε έτος, μέχρι να φτάσει να έχει λιγότερες από 365 ή 366 μέρες. Τότε βρίσκει το μήνα με παρόμοιο τρόπο και οι υπόλοιπες μέρες που μένουν είναι η μέρα του μήνα. Από τα δευτερόλεπτα που δεν φτάνουν να συμπληρώσουν μία ημέρα (t % DAYSEC), με μία διαίρεση είναι απλή υπόθεση να βρεθεί η ώρα, το λεπτό και το δευτερόλεπτο.

Οι υπόλοιπες συναρτήσεις στο klibc/time.c είναι πολύ απλές. Η time προσθέτει τα δευτερόλεπτα που έχουν περάσει από την εκκίνηση του πυρήνα με τα δευτερόλεπτα που έχουμε μετρήσει με τον timer, για να μας επιστρέψει δευτερόλεπτα από το epoch, ενώ η asctime δέχεται ένα struct tm, και μας επιστρέφει ένα string με την ημερομηνία και ώρα σε human–readable μορφή.



1 **Αποτέλεσμα της εκτέλεσης του kernel.**

> **Δίσεκτα κατά το Γρηγοριανό ημερολόγιο είναι όλα τα χρόνια που διαιρούνται ακριβώς με το 4 ή με το 400, εξαιρουμένων αυτών που διαιρούνται ακριβώς με το 100.**

Δοκιμές με το χρόνο

Για να βεβαιωθούμε ότι δουλεύουν όλα τα παραπάνω, κατ' αρχάς η init_rtc τυπώνει την αρχική ημερομηνία και την ώρα που παίρνει διαβάζοντας το RTC. Επιπλέον, τροποποιούμε τον timer interrupt handler που είδαμε παραπάνω, ώστε σε κάθε δευτερόλεπτο που περνά, να μας λέει την παρούσα ημερομηνία και ώρα.

```
static void intr_handler()
{
    if(++nticks % TICK_FREQ_HZ == 0) {
        time_t t = time(0);
        printf("%s", asctime(gmtime(&t)));
    }
}
```

Αν όλα βαίνουν καλώς, πρέπει να δούμε κάτι σαν αυτό που δείχνει η **εικόνα 1.**

Ασκήσεις για τον αναγνώστη

Είπαμε προηγουμένως ότι ο counter 2 του timer είναι συνδεδεμένος στο PC speaker, για να μας επιτρέπει να δημιουργούμε ήχους σε διάφορες συχνότητες. Δοκιμάστε να υλοποιήσετε μία συνάρτηση beep(int frequency, int duration), και χρησιμοποιήστε τη για να παίξετε μία μικρή μελωδία από διαφορετικούς τόνους. Πέρα από τον προγραμματισμό του timer που εξηγήσαμε παραπάνω, θα χρειαστεί να θέσετε τα bits 0 και 1 στο port 0x61 του keyboard controller για να αρχίσει να βγάζει ήχο το speaker, ενώ μηδενίζοντάς τα, σταματά.