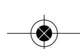


Του Γιάννη Τσιομπίκα <nuclear@member.fsf.org>

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

Εισαγωγή στο kernel development

Το δεύτερο μέρος της σειράς προγραμματισμού πυρήνων από το μηδέν μπαίνει βαθύτερα στην αρχιτεκτονική του x86, εξερευνώντας το μοντέλο μνήμης του επεξεργαστή και το χειρισμό των interrupts.

Για Smartphones

Εργαλεία: GCC, GNU make, GRUB, qemu
Δυσκολία: ★★★★★
URL: http://goo.gl/sfjx2

Στο προηγούμενο τεύχος είδαμε τη διαδικασία του boot και πώς να κάνουμε τον kernel μας να συμβατός με multiboot bootloaders, όπως το GRUB. Επίσης, είδαμε πώς μπορούμε να χρησιμοποιήσουμε τη VGA σε text mode για να βγάλουμε κείμενο στην οθόνη. Εάν χάσατε το προηγούμενο τεύχος, μην πανικοβάλλεστε, τα παλαιότερα άρθρα, όπως και ο συνοδευτικός κώδικας κάθε άρθρου, είναι διαθέσιμα από το URL που δίνουμε στην κορυφή της πρώτης σελίδας. Στο συνοδευτικό κώδικα αυτού του τεύχους συμπεριλαμβάνονται οι λύσεις στις «ασκήσεις για τους αναγνώστες» του προηγούμενου (printf implementation και VGA hardware scrolling), καθώς, φυσικά, και όλες οι καινούργιες προσθήκες στον kernel μας, που θα συζητήσουμε σε αυτό το τεύχος.

Μοντέλο μνήμης σε Protected Mode

Η αρχιτεκτονική του x86 είναι βασισμένη στην ιδέα ότι η μνήμη είναι χωρισμένη σε τμήματα (segments). Κάθε πρόσβαση στη μνήμη χρησιμοποιεί έναν segment selector, που διαλέγει ένα από τα segments της μνήμης, καθώς και ένα offset που είναι η απόσταση από την αρχή του επιλεγμένου segment. Αυτά τα δύο στοιχεία αποτελούν αυτό που ονομάζεται «λογική διεύθυνση» και ο επεξεργαστής πρέπει να τη μετατρέψει σε φυσική διεύθυνση μνήμης, προτού γίνει η πρόσβαση.

Ο selector είναι ουσιαστικά ένα index στον πίνακα με τα segments που έχουμε ορίσει (βλ. συνάρτηση selector στο αρχείο segm.c). Ο πίνακας αυτός λέγεται Global Descriptor Table (GDT) και περιέχει μία σειρά από πεδία με συγκεκριμένη δομή, τα οποία ονομάζονται segment descriptors και περιγράφουν τα πάντα για ένα segment: Από πού ξεκινάει (segment base), πόσο μέγεθος έχει (segment limit), το privilege level του, που έχει να κάνει με το αν ανήκει στον kernel ή σε κάποιο user process, αν περιέχει δεδομένα η εκτελέσιμο κώδικα, και διάφορα άλλα.

Οταν, λοιπόν, ο επεξεργαστής θέλει να μεταφράσει μία λογική διεύθυνση σε φυσική διεύθυνση μνήμης, θεωρώντας ότι δεν έχουμε paging ενεργοποιημένο, που θα το δούμε σε επόμενο άρθρο, παίρνει την αρχή του segment, προσθέτει το offset και έτσι αποκτά τη διεύθυνση που μπορεί να χρησιμοποιηθεί, για να γίνει η πρόσβαση στη μνήμη του συστήματος. Ενα τέτοιο segmented μοντέλο μνήμης θα μπορούσε να χρησιμοποιηθεί για να σπάσουμε τη διαθέσιμη μνήμη σε κομμάτια και να δώσουμε ένα κομμάτι σε κάθε process.

Ο επεξεργαστής ελέγχει και απαγορεύει την πρόσβαση σε μνήμη έξω από το ενεργό segment και έτσι μπορούμε να είμαστε σίγουροι ότι κανένα process δεν θα πειράξει τη μνήμη που έχει ανατεθεί σε άλλα ή στο λειτουργικό σύστημα. Αυτό το μοντέλο ταιριάζει τέλεια σε ένα batch processing σύστημα του 1960 και κάτι παρεμφερές χρησιμοποιούσαν και οι πρώτες εκδόσεις του UNIX που έτρεχαν σε PDP11 χωρίς virtual memory, αλλά δεν μας βοηθά, εάν ο στόχος μας είναι να φτιάξουμε έναν σύγχρονο virtual memory kernel.

Το segmentation στον x86 δεν μπορεί να απενεργοποιηθεί, οπότε πρέπει να βρούμε έναν τρόπο για να τον βγάλουμε από τη μέση και να απλοποιήσουμε το μοντέλο της μνήμης, κάτι που ευτυχώς γίνεται εύκολα.

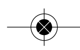
Ο τρόπος που μπορούμε να το πετύχουμε αυτό και να αποκτήσουμε έναν ενιαίο χώρο διευθύνσεων, ώστε να σταματήσουμε να ασχολούμαστε με διαφορετικά segments, να αφήσουμε τα selectors στην ησυχία τους και να πετύχουμε οι λογικές διευθύνσεις να γίνονται map μία προς μία με τις φυσικές, είναι να κάνουμε όλα τα segments να αλληλοκαλύπτονται, θέτοντας για όλα αρχική διεύθυνση το 0 και όριο 4gb, που είναι το μέγιστο δυνατό.

Ανά πάσα στιγμή είναι ενεργά τουλάχιστον δύο segments, ένα για κώδικα και ένα για δεδομένα. Οταν ο επεξεργαστής πάει να τραβήξει εντολές (κώδικα) από τη μνήμη, χρησιμοποιεί τον CS selector (και, φυσικά, τα περιεχόμενα του EIP σαν offset). Ενώ όταν διαβάζει ή γράφει δεδομένα στη μνήμη, χρησιμοποιεί συνήθως τον DS selector, εκτός από μερικές εντολές που χρησιμοποιούν τον SS selector ή τον ES selector εξ ορισμού.

Οπότε, αρκεί για αρχή να φτιάξουμε δύο segment descriptors και να τα τοποθετήσουμε στον GDT. Αυτό γίνεται στη συνάρτηση init_segm στο αρχείο segm.c, η οποία καλείται στην αρχή της kmain. Το descriptor στη θέση 0 του GDT δεν χρησιμοποιείται, οπότε το αφήνουμε μηδενισμένο, ενώ στις άλλες δύο θέσεις φτιάχνουμε kernel code και kernel data segment descriptors που περιγράφουν δύο επικαλυπτόμενα segments με αρχή το 0 και όριο 0xffffffff. Κατόπιν, καλούμε τη set_gdt, η οποία χρησιμοποιεί την εντολή lgdt για να φορτώσει τη διεύθυνση και το μέγεθος του καινούργιου GDT στον καταχωρητή gdtr, ώστε να αρχίσει να τον χρησιμοποιεί ο επεξεργαστής. Τέλος, ενεργοποιούμε τα καινούργια segments, τοποθετώντας τους κατάλληλους selectors σε όλα τα segment registers, με τη συνάρτηση setup_selectors.

Κώδικας: segm-asm.S

```
    .data
    .align 4
off:.long 0  /* memory for setup_selectors */
seg:.short 0
lim:.short 0 /* memory for set_gdt */
addr:.long 0

    .text
/* setup_selectors(uint16_t code, uint16_t data) */
```

```
    .globl setup_selectors
setup_selectors:
    /* set data selectors directly */
    movl 8(%esp), %eax
    movw %ax, %ss
    movw %ax, %es
    movw %ax, %ds
    movw %ax, %gs
    movw %ax, %fs
    /* set cs using a long jump */
    movl 4(%esp), %eax
    movw %ax, (seg)
    movl $ldcs, (off)
    ljmp *off
ldcs:
    ret
/* set_gdt(uint32_t addr, uint16_t limit) */
    .globl set_gdt
set_gdt:
    movl 4(%esp), %eax
    movl %eax, (addr)
    movw 8(%esp), %ax
    movw %ax, (lim)
    lgdt (lim)
    ret
```

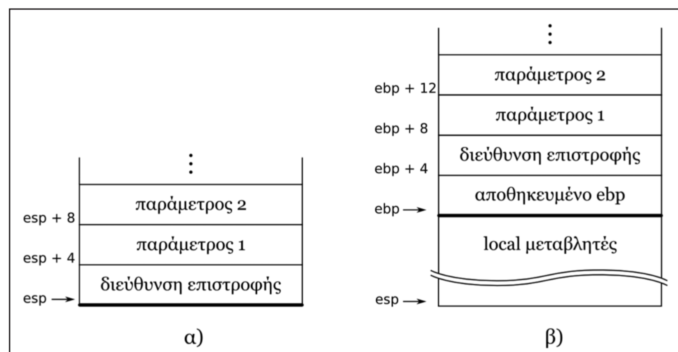
Calling conventions

Τις παραπάνω συναρτήσεις τις καλούμε από C κώδικα και πρέπει να ακολουθήσουμε τις ίδιες συμβάσεις που ακολουθεί και ο C compiler για να μπορέσουμε να πάρουμε τις παραμέτρους που μας περνά ο κώδικας που κάλεσε τη συνάρτηση, αλλά και να μην καταστρέψουμε δεδομένα που ο compiler θεωρεί ότι έχουν διάφοροι καταχωρητές. Εν ολίγοις, πρέπει να προσέξουμε να μην πειράξουμε τους καταχωρητές ebx, esi, edi, ebp, ds, es, και ss (ή, αν χρειαστεί να τους χρησιμοποιήσουμε, πρέπει να φροντίσουμε να τους επαναφέρουμε στην αρχική κατάστασή τους προτού επιστρέψουμε από τη συνάρτηση). Τις παραμέτρους τις κάνει push με αντίστροφη σειρά στο stack ο κώδικας που μας καλεί και τις κάνει pop αφού επιστρέψουμε. Τέλος, ό,τι χρειάζεται να επιστρέψουμε, το αφήνουμε στον eax προτού κάνουμε ret.

Από τα παραπάνω βγαίνει το συμπέρασμα ότι μπορούμε να διαβάσουμε τις παραμέτρους χρησιμοποιώντας offsets από τον stack pointer (esp).

Οταν αρχίζει να εκτελείται η συνάρτηση, ο esp δείχνει στο τελευταίο πράγμα που έγινε push, το οποίο είναι η διεύθυνση επιστροφής που κάνει αυτόματα push η εντολή call. Αντίστροφα, η εντολή ret αντιγράφει στον eip τη διεύθυνση επιστροφής που βρίσκει στο stack, ώστε να επιστρέψει η εκτέλεση του προγράμματος εκεί όπου έγινε η κλήση. Ετσι, αν προσθέσουμε 4 στη διεύθυνση που περιέχει ο esp, θα πάρουμε pointer στην πρώτη παράμετρο της συνάρτησής μας, η οποία ήταν το τελευταίο πράγμα που έγινε push πριν από την εκτέλεση της call. Αν προσθέσουμε 8, θα πάρουμε pointer στη δεύτερη παράμετρο κ.ο.κ. Το **σχήμα 1α** δείχνει πώς ακριβώς είναι το stack όταν αρχίζει η εκτέλεση μίας συνάρτησης με δύο παραμέτρους.

Εάν χρειαστεί να δημιουργήσουμε χώρο στο stack για local μεταβλητές ή για να σώσουμε registers ώστε να τους επαναφέρουμε αργότερα, τότε θα πρέπει να τροποποιήσουμε τον esp. Σε αυτή την περίπτωση, για να μπορούμε να βρούμε τις παραμέτρους μας, σώζουμε την προηγούμενη τιμή του ebp,



1 Η κατάσταση του stack όταν αρχίζει η εκτέλεση μίας συνάρτησης με δύο παραμέτρους: α) Χωρίς χώρο για local μεταβλητές. β) Με χώρο για local μεταβλητές.

κάνοντάς την push στο stack, και κρατάμε στον ebp την τιμή του esp προτού τον πειράξουμε για να δημιουργήσουμε χώρο. Ετσι, πλέον χρησιμοποιούμε τον ebp για να βρούμε σε θετικά offsets τις παραμέτρους (αρχίζοντας από το offset 8 αυτή τη φορά), και αρνητικά για τις local μεταβλητές **(βλ. σχήμα 1β).**

Τα segment descriptors που τοποθετούμε στον GDT είναι 8 bytes καθένα και έχουν μία πολύ συγκεκριμένη και αρκετά περίπλοκη δομή, την οποία περιμένει να δει ο επεξεργαστής. Δεν θα μπούμε σε λεπτομερή περιγραφή καθενός bit που τοποθετούμε εκεί, μια και αυτή η δομή περιγράφεται λεπτομερώς στα specifications της Intel **(βλ. σχήμα 2α)**. Στον κώδικά μας, το descriptor το αναπαριστά ένα struct που περιέχει έναν πίνακα με 4 16bit unsigned integers (δηλωμένο στο desc.h), και έχουμε μία συνάρτηση, τη segm_desc (ορισμένη στο segm.c) που κατασκευάζει ένα descriptor, τοποθετώντας όλα τα πεδία στις κατάλληλες θέσεις.

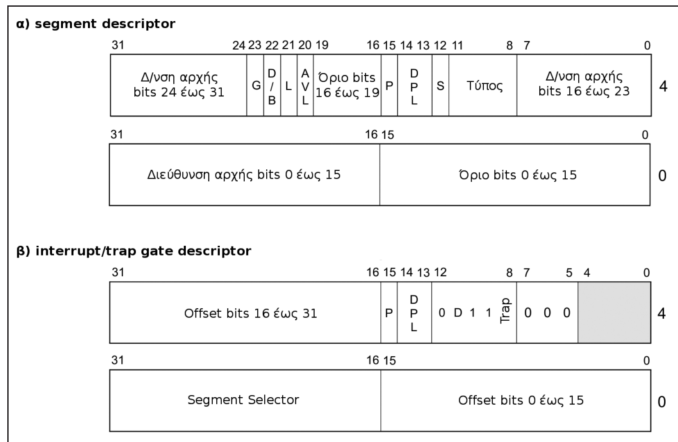
```
Κώδικας segm_desc :
void segm_desc(desc_t *desc, uint32_t base, uint32_t
limit, int dpl, int type)
{
    uint16_t flags;

    desc->d[0] = limit & 0xffff;
    desc->d[1] = base & 0xffff;

    flags = BIT_PRESENT | BIT_NOSYS | BIT_WR;
    if(type == TYPE_DATA) {
        flags |= BIT_CODE;
    }
    desc->d[2] = ((base >> 16) & 0xff) |
        ((dpl & 3) << 13) | flags;
    desc->d[3] = ((limit >> 16) & 0xf) | BIT_GRAN |
        BIT_BIG | ((base >> 16) & 0xff00);
}
```

Interrupts

Ο χειρισμός των interrupts είναι μία από τις πιο βασικές λειτουργίες ενός πυρήνα, μια και ουσιαστικά κάθε φορά που εκτελείται κώδικας του πυρήνα μετά το initialization, αυτό είναι σαν απάντηση σε ένα interrupt. Τα interrupts διακόπτουν την κανονική ροή εκτέλεσης, για να χειριστούμε κάποιο γεγονός που συνέβη.

Υπάρχουν τριών ειδών interrupts: Hardware, software, και exceptions. Τα hardware interrupts ειδοποιούν τον επεξεργαστή ότι μία συσκευή του συστήματος χρειάζεται την προσοχή

LINUX LABS KERNEL_PRODUCTIVITY 3/8/11 3:41 PM Page 86



2 **Η δομή των α) segment descriptors στο GDT και β) interrupt gate και trap gate descriptors στο IDT**

μας, π.χ., όταν ο χρήστης πατήσει κάποιο πλήκτρο, όταν ο δίσκος τελειώσει την ανάγνωση δεδομένων που ζητήσαμε κ.λπ. Τα exceptions είναι μηχανισμός όπου ο ίδιος ο επεξεργαστής διακόπτει τη ροή εκτέλεσης εξαιτίας κάποιου προβλήματος που προέκυψε από την εκτέλεση των εντολών του προγράμματος, π.χ., διαίρεση διά του 0 ή πρόσβαση σε μνήμη έξω από τα όρια του segment. Τέλος, software interrupts δημιουργούνται με την εντολή int, κάτι που είναι πολύ χρήσιμο για να μπορεί κάποιο user process να ζητήσει κάτι από τον πυρήνα (system call).

Οταν «σηκωθεί» ένα interrupt, ο επεξεργαστής σταματάει να εκτελεί εντολές σειριακά, αποθηκεύει τα περιεχόμενα του instruction pointer (eip) στο stack μαζί με μερικές άλλες πληροφορίες και συνεχίζει να εκτελεί κώδικα από ένα σημείο στη μνήμη που έχει οριστεί ότι χειρίζεται το συγκεκριμένο interrupt. Οταν τελειώσει η εκτέλεση του interrupt handler η εντολή iret κάνει τον επεξεργαστή να αντιγράψει στον eip την προηγούμενη αποθηκευμένη τιμή του από το stack, ώστε να συνεχιστεί η εκτέλεση από το σημείο που σταμάτησε προτού σηκωθεί το interrupt.

Η παραπάνω περιγραφή είναι υπεραπλουστευμένη, μια και ο επεξεργαστής, αναλόγως αν εκτελούσε κώδικα σε user privilege level (3) η kernel privilege level (0) προτού σηκωθεί το interrupt, μπορεί να κάνει μία περισσότερο η λιγότερο περίπλοκη διαδικασία για να αρχίσει να εκτελεί τον interrupt handler που είναι πάντα σε kernel code segment (privilege level 0). Επίσης, σε γενικές γραμμές προτού επιστρέψει η εκτέλεση σε user κώδικα, όταν τελειώσει τη δουλειά του ο interrupt handler, ο πυρήνας μπορεί να κάνει τα μαγικά του, ώστε η εκτέλεση να συνεχίσει σε άλλο process από αυτό που εκτελούνταν προηγουμένως, αλλά όλα αυτά θα τα δούμε σε μεταγενέστερο άρθρο στο οποίο θα υλοποιήσουμε processes και scheduling.

Στον x86 κάθε interrupt έχει έναν αριθμό από το 0 έως το 255. Τα interrupts 0 έως και 31 είναι δεσμευμένα για exceptions του επεξεργαστή, ενώ τα υπόλοιπα μπορούν να χρησιμοποιηθούν για hardware και software interrupts. Ο επεξεργαστής , όταν σηκωθεί ένα interrupt, για να βρει πού να ανακατευθύνει την εκτέλεση, χρησιμοποιεί αυτόν τον αριθμό ως index σε έναν πίνακα με interrupt gate descriptors ή trap gate descriptors, που λέγεται Interrupt Descriptor Table (IDT). Κάθε gate descriptor περιέχει τη λογική διεύθυνση (segment selector και offset) του σημείου στη μνήμη (entry point) όπου ξεκινά ο κώδικας που θα χειριστεί το συγκεκριμένο interrupt.

Στον κώδικά μας η συνάρτηση gate_desc, στο αρχείο intr.c κατασκευάζει ένα gate descriptor, η οποία είναι πολύ παρόμοια με τη segm_desc που είδαμε παραπάνω. Επιπλέον, έχουμε και τη βοηθητική συνάρτηση set_intr_entry, που παίρνει έναν αριθμό interrupt και ένα function pointer και καλεί την gate_desc για να δημιουργήσει το κατάλληλο descriptor και να το τοποθετήσει στον IDT.

Χειρισμός interrupts

Το entry point για τον κώδικα που θα χειριστεί τα interrupts, είναι απαραίτητο να γραφτεί σε assembly για δύο λόγους. Κατ' αρχάς, το stack frame όταν μπαίνουμε στο interrupt είναι διαφορετικό από το stack frame κατά την κλήση C συναρτήσεων και μάλιστα διαφέρει από interrupt σε interrupt, αφού σε κάποια exceptions ο επεξεργαστής κάνει push ένα error code στο stack, ενώ σε άλλα exceptions και σε όλα τα υπόλοιπα interrupts όχι.

Κατά δεύτερον, πρέπει να εκτελέσουμε συγκεκριμένες εντολές, όπως η iret για επιστροφή από το interrupt, αντί για τη ret που παράγει κανονικά ο compiler για επιστροφή από συνάρτηση, και τις pusha/popa για να αποθηκεύσουμε όλους τους registers στο stack, μπαίνοντας στο interrupt και να τους επαναφέρουμε προτού επιστρέψουμε, ώστε να μπορεί να συνεχίσει κανονικά η εκτέλεση του κώδικα από εκεί όπου σταμάτησε.

Για τους παραπάνω λόγους, στο συνοδευτικό κώδικα έχουμε σπάσει το χειρισμό interrupts σε δύο κομμάτια, το assembly entry point στο intr-asm.S και τη C συνάρτηση dispatch_intr στο intr.c.

Η εκτέλεση ξεκινάει σε μία σειρά από assembly συναρτήσεις, μία για κάθε πιθανό interrupt, που έχουν δύο μορφές: Αυτές που χρησιμοποιούνται για exceptions με error code στο stack και αυτές που χρησιμοποιούνται σε όλες τις υπόλοιπες περιπτώσεις.

Αυτές με το error code κάνουν push στο stack τον αριθμό του interrupt και κάνουν jump στο label intr_entry_common. Αυτές χωρίς error code κάνουν πρώτα push ένα ψεύτικο error code (την τιμή 0) και μετά τον αριθμό του interrupt προτού κάνουν jump στο intr_entry_common, έτσι ώστε και στις δύο περιπτώσεις το stack frame να είναι ίδιο από αυτό το σημείο και μετά.

Ο κώδικας στο intr_entry_common κάνει pusha για να αποθηκεύσει όλους τους registers στο stack και καλεί την dispatch_intr. Οταν επιστρέψει η dispatch_intr, επαναφέρουμε όλους τους registers στην προηγούμενη κατάστασή τους με ένα popa, και καθαρίζουμε τις δύο τιμές (error code και αριθμός interrupt) από το stack απλώς αυξάνοντας τον esp κατά 8 προτού κάνουμε iret.

Ο **Alan Turing** σοφά είχε πει ότι ο προγραμματισμός δεν κινδυνεύει να γίνει ποτέ βαρετή, μηχανική, διαδικασία, γιατί όλες τις μηχανικές δουλειές μπορούμε να τις αφήσουμε για τον ίδιο τον υπολογιστή.

Ομως, η παραπάνω διαδικασία, το να γράψουμε, δηλαδή, όλα αυτά τα διαφορετικά αντίγραφα του entry για κάθε interrupt είναι απελπιστικά βαρετή και μηχανική, άρα, αν ξεκινήσουμε να την κάνουμε με το χέρι, κάτι κάνουμε λάθος. Είναι καλύτερο να επιστρατεύσουμε τα macros του GNU assembler και να τον αφήσουμε να κάνει τη δουλειά για εμάς.

Αρκεί να γράψουμε μόνο δύο macros για τις δύο περιπτώσεις που περιγράψαμε παραπάνω, δηλαδή, για interrupts που ξεκινάνε με ή χωρίς error code στο stack, και

μετά να γράψουμε μία γραμμή για κάθε πιθανό interrupt που θα καλεί αυτά τα macros με τις σωστές παραμέτρους.

Αλλά, γιατί να σταματήσουμε εκεί; Και στο C κώδικα, στη συνάρτηση init_intr που καλείται στην αρχή για να αρχικοποιήσουμε το χειρισμό interrupts του πυρήνα, θα χρειαστεί να καλέσουμε τη συνάρτηση set_intr_entry για καθένα από τα interrupts, ώστε να γεμίσουμε το IDT με τα κατάλληλα descriptors που δείχνουν στα entry points που δημιουργήσαμε με αυτά τα macros.

Μπορούμε να χρησιμοποιήσουμε τα ίδια macro invocations, ώστε στη μία περίπτωση να δημιουργηθεί ο κατάλληλος assembly κώδικας και στην άλλη περίπτωση C κώδικας; Η απάντηση, φυσικά, είναι ναι, με συνδυασμό GNU assembler macros και C preprocessor macros και includes.

Στην init_intr, αφού καλέσουμε τη set_idt για να πούμε στον επεξεργαστή πού ακριβώς βρίσκεται ο πίνακας με τα descriptors, κάνουμε include το αρχείο "interrupts.h". Αυτό έχει ως αποτέλεσμα να μπουν σε αυτό το σημείο όλες οι κλήσεις στα C macros INTR_ENTRY_EC και INTR_ENTRY_NOEC, τα οποία, αφού δεν είναι defined το ASM, κάνουν expand σε ένα function prototype για κάθε entry point και μία κλήση στη set_intr_entry με τον αριθμό και το όνομα του entry point.

Αντίθετα, στο τέλος του αρχείου intr-asm.S, όπου έχουμε κάνει πρώτα define το ASM, οι κλήσεις στα C macros INTR_ENTRY_EC και INTR_ENTRY_NOEC γίνονται expand σε κλήσεις των GNU assembler macros ientry_err και ientry_noerr, τα οποία είναι δηλωμένα όπως περιγράψαμε παραπάνω.

Κώδικας intr-asm.S:

```
/* interrupt entry with error code macro */
    .macro ientry_err n name
    .globl intr_entry_\name
intr_entry_\name:
    pushl $\n
    jmp intr_entry_common
    .endm

/* interrupt entry without error code macro */
    .macro ientry_noerr n name
    .globl intr_entry_\name
intr_entry_\name:
    pushl $0
    pushl $\n
    jmp intr_entry_common
    .endm

/* common code used by all entry points */
    .extern dispatch_intr
intr_entry_common:
    pusha
    call dispatch_intr
    popa
    /* remove error code and intr num from stack */
    add $8, %esp
    iret

#define ASM
#include <interrupts.h>
```

Στη C πλευρά, η dispatch_intr απλώς κοιτάει σε ένα array από function pointers, αν έχει οριστεί handler συνάρτηση για το συγκεκριμένο interrupt και, αν υπάρχει, την καλεί. Αν δεν υπάρχει, απλώς κάνει output ένα μήνυμα "unhandled interrupt" με τον αριθμό του. Η συνάρτηση "interrupt" επιτρέπει σε οποιοδήποτε υποσύστημα του πυρήνα να θέσει handler function σε αυτόν τον πίνακα, για όποιο interrupt θέλει να χειρίζεται.

Το μόνο σημείο της dispatch_intr που χρειάζεται επεξήγηση, είναι το πώς ξέρει το interrupt number και το exception error code. Οπως είδαμε παραπάνω, ο αριθμός του interrupt γίνεται push στο stack, μαζί με όλους τους registers και το error code ή ένα ψεύτικο error code, αν αυτό δεν υπάρχει.

Αυτό σημαίνει ότι όλα αυτά μπορούμε στο C κώδικα να τα θεωρήσουμε ως παραμέτρους της συνάρτησης, μια και οι παράμετροι περνάνε με τον ίδιο ακριβώς τρόπο όπως είπαμε νωρίτερα. Αυτό ακριβώς κάνει ο κώδικας στο intr.c, απλώς, αντί να δηλώσουμε τη συνάρτηση με δεκάδες παραμέτρους για όλα όσα έχουν γίνει push στο stack, θεωρούμε ότι όλα αυτά είναι μέρος ενός μεγάλου struct και ορίζουμε τη συνάρτηση σαν να έχει αυτό το struct ως μοναδική παράμετρο.

IRQs

Κάθε συσκευή στο PC που πρέπει να μπορεί να μας ειδοποιεί ασύγχρονα για κάτι, παίρνει και μία γραμμή IRQ (interrupt request) στο intel 8259 PIC chip (Programmable Interrupt Controller), η δουλειά του οποίου είναι να σηκώνει interrupt στον επεξεργαστή όταν δεχτεί κάποιο IRQ σήμα και να ανακοινώνει τον αριθμό του interrupt, τοποθετώντας τον στο data bus.

Ο PIC έχει μόλις οκτώ IRQ lines, αλλά στο PC υπάρχουν δύο τέτοια chips συνδεδεμένα σε master/slave διαρρύθμιση, ώστε να έχουμε 15 IRQs διαθέσιμα για συσκευές.

Προτού αρχίσει να διαχειρίζεται IRQs κάθε PIC, πρέπει να τον κάνουμε initialize, το οποίο αναλαμβάνει η συνάρτηση init_pic, στέλνοντας μία σειρά από commands στα ports των δύο PIC. Μέρος του initialization είναι να του δώσουμε ένα offset, το οποίο θα προσθέτει στο νούμερο του interrupt που ανακοινώνει στον επεξεργαστή. Αυτό είναι σημαντικό, γιατί τα πρώτα 32 interrupts, όπως είπαμε, είναι δεσμευμένα για exceptions και άρα δεν μπορούν να χρησιμοποιηθούν. Το offset που θα χρησιμοποιήσουμε στον πυρήνα μας, ορίζεται από το define IRQ_OFFSET σε 32, έτσι, όταν ο PIC λαμβάνει IRQ 0 (το οποίο αντιστοιχεί στον timer του συστήματος), εμείς θα παίρνουμε interrupt 32, όταν λαμβάνει IRQ 1 (keyboard), εμείς θα παίρνουμε interrupt 33 κ.ο.κ.

Οταν τελειώσουμε με το χειρισμό κάποιου interrupt που έχει προέλθει από τον PIC, πρέπει να του στείλουμε ένα end of interrupt command, το οποίο αναλαμβάνει η συνάρτηση end_of_irq, που καλείται στο τέλος της dispatch_intr για interrupts 32 έως 47.

Δοκιμάζοντας τα interrupts

Το τελευταίο πράγμα που κάνει η init_intr, είναι να εκτελέσει την εντολή sti μέσω του macro enable_intr, για να αρχίσει να δέχεται interrupts ο επεξεργαστής. Σε αυτή τη φάση, αν τρέξουμε τον kernel, θα δούμε ότι αρχίζει μία ασταμάτητη ροή από μηνύματα "unhandled interrupt 32", που είναι το IRQ 0 που ενεργοποιείται συνεχώς με συγκεκριμένη συχνότητα από τον timer του υπολογιστή.

Την επόμενη φορά θα δούμε πώς μπορούμε να χρησιμοποιήσουμε τα interrupts για να κάνουμε χρήσιμα πράγματα, π.χ., για να μετράμε το χρόνο ή να πάρουμε είσοδο από το πληκτρολόγιο. •