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

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

Εισαγωγή στο kernel development

Συνεχίζουμε το ταξίδι στη δημιουργία ενός νέου, ολόδικού μας πυρήνα. Αυτή τη φορά, θα δούμε πώς γίνεται το paging και η διαχείριση της μνήμης.

Για Smartphones

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

Στο προηγούμενο άρθρο είδαμε το μοντέλο μνήμης του x86 και βγάλαμε το **segmentation** από τη μέση, δημιουργώντας όλα τα απαραίτητα segments αλληλοεπικαλυπτόμενα με αρχή το 0 και μέγεθος όσο όλο το **address space (4GB)**. Αυτό το κάναμε, επειδή στόχος μας είναι να δημιουργήσουμε ένα μοντέρνο πυρήνα με εικονική μνήμη.

Σε αυτό το τεύχος θα ασχοληθούμε με το **paging** και τη **διαχείριση της μνήμης**, κάτι απαραίτητο ώστε όλα τα κομμάτια του πυρήνα που θα γράψουμε μετά να μπορούν να κάνουν allocate μνήμη. Μην ξεχάσετε να κατεβάσετε το συνοδευτικό κώδικα από το Web site των άρθρων που δίνεται στην κορυφή της σελίδας. Και όπως πάντα, αν χάσατε κάποιο από τα προηγούμενα άρθρα, θα τα βρείτε στο ίδιο site υπό τους όρους της άδειας Creative Commons Attribution-ShareAlike.

Concurrency

Προτού μπούμε στο ψητό, πρέπει να αναφερθούμε εν συντομία στο θέμα του **concurrency**. Μετά την ενεργοποίηση των interrupts στο προηγούμενο άρθρο, κομμάτια του κώδικα του πυρήνα μπορεί να εκτελεστούν οποιαδήποτε στιγμή ασύγχρονα. Αυτό σημαίνει ότι πρέπει να προσέξουμε να προστατεύσουμε τα critical sections του πυρήνα με κάποιου είδους αμοιβαίο αποκλεισμό, ώστε να μη γίνεται ταυτόχρονη πρόσβαση σε κοινές δομές δεδομένων του πυρήνα. Ο πιο απλός τρόπος να το πετύχουμε αυτό, είναι να φροντίσουμε να απενεργοποιούμε τα interrupts, μπαίνοντας στο critical section, ώστε να είμαστε σίγουροι ότι δεν θα διακοπεί η εκτέλεση για να τρέξει κάτι άλλο, και να τα επαναφέρουμε στην αρχική κατάστασή τους όταν τελειώσουμε. Αυτό το πετυχαίνει το επόμενο απόσπασμα, καλώντας τις **get_intr_state και set_intr_state** που είναι ορισμένες στο intr-asm.S.

```
int istate = get_intr_state();
disable_intr();
/* ... critical section ... */
set_intr_state(istate);
```

Διαχείριση φυσικής μνήμης

Προτού αρχίσουμε με τη διαχείριση και τη διανομή της μνήμης, πρέπει, φυσικά, να ξέρουμε πόση μνήμη έχουμε εγκατεστημένη στο σύστημα και ποια τμήματα φυσικών διευθύνσεων μπορούμε να χρησιμοποιήσουμε. Πληροφορίες για τη διαθέσιμη μνήμη κανονικά παίρνουμε με κλήση στο **BIOS**, το οποίο, όμως, είναι αρκετά δύσκολο να γίνει από protected mode, μια και οι κλήσεις του BIOS γίνονται με **real mode interrupts.**

Ευτυχώς το **multiboot standard** προέβλεψε αυτή τη δυσκολία και έτσι ο boot loader μάς παρέχει πληροφορίες για τη διαθέσιμη μνήμη του συστήματος σε εύχρηστη μορφή. Οταν ο boot loader περνά τον έλεγχο στο entry point του kernel, μας αφήνει στον **ebx register** τη διεύθυνση στην οποία έχει τοποθετήσει μία δομή (βλ. **mboot_info** στο mboot.h) με διάφορες πληροφορίες που του έχουμε ζητήσει, θέτοντας συγκεκριμένα bits στο πεδίο flags του multiboot header. Αυτό το struct, το οποίο περνάμε πλέον ως παράμετρο στην **kmain** και αυτή με τη σειρά της στην **init_mem**, περιέχει έναν pointer σε έναν πίνακα με διαθέσιμα και μη τμήματα διευθύνσεων. Η συνάρτηση **init_mem** στο αρχείο mem.c διαβάζει ένα–ένα τα διαστήματα από αυτόν τον πίνακα και για κάθε διαθέσιμο διάστημα καλεί την **add_memory** για να το προσθέσει στα κομμάτια μνήμης που θα διαχειρίζεται ο πυρήνας.

Η διαχείριση φυσικής μνήμης γίνεται σε επίπεδο page (δηλαδή, κομμάτια των 4KB). Οι δομές που χρησιμοποιεί ο **physical page manager**, πρέπει να είναι όσο πιο απλές γίνεται, και στατικές, ώστε να μη χρειάζεται δυναμική δέσμευση μνήμης, μια και προφανώς δεν μπορούμε να κάνουμε allocate μνήμη χωρίς allocator. Γι' αυτόν το λόγο, χρησιμοποιούμε ένα **bitmap** για να σημαδέψουμε ποια pages είναι ελεύθερα και ποια όχι. Το bitmap αυτό, το οποίο βάζουμε αυθαίρετα να ξεκινάει από το τέλος του **kernel image** που σηματοδοτείται από το σύμβολο _end, είναι απλώς ένα κομμάτι μνήμης του οποίου κάθε bit αντιστοιχεί σε ένα page φυσικής μνήμης.

Για να διαπιστώσουμε αν το 23ο page είναι ελεύθερο, αρκεί να κοιτάξουμε αν το 23ο κατά σειρά bit είναι 0 ή 1. Στην πράξη, θέτουμε ένα **uint32_t pointer** στην αρχή του bitmap, ώστε να το βλέπουμε σαν **array** από 32bit ακέραιους, και όταν ψάχνουμε για ελεύθερα pages να διαβάζουμε 32 bits κάθε φορά. Αν η τιμή που θα δούμε είναι **ffffffff**, τότε ξέρουμε ότι δεν υπάρχει κανένα ελεύθερο ανάμεσα στα 32 pages που αντιστοιχούν σε αυτό το σημείο του bitmap και μπορούμε να πάμε στο επόμενο που αντιστοιχεί στα επόμενα 32 pages. Μόλις βρούμε ένα κομμάτι του bitmap με διαφορετική τιμή, ξέρουμε ότι εδώ υπάρχει ελεύθερο page και ψάχνουμε ένα–ένα τα bits με shifting και masking, ώστε να βρούμε ποιο είναι το ελεύθερο page.

Ολες οι αλλαγές στο bitmap γίνονται από τη συνάρτηση mark_page που βρίσκει και θέτει το σωστό bit 0 ή 1 για να σημειωθεί κάποιο page ως ελεύθερο η δεσμευμένο.

Η **add_memory** που προαναφέραμε, καλεί τη mark_page για να σημειώσει ως ελεύθερα όλα τα pages που αντιστοιχούν στα διαθέσιμα κομμάτια μνήμης που μας έδωσε ο boot loader. Τέλος, η **alloc_phys_page** υλοποιεί τον αλγόριθμο αναζήτησης ελεύθερων pages που μόλις αναφέραμε, η οποία μας επιστρέφει τη διεύθυνση από την οποία ξεκινά το page που έκανε allocate.

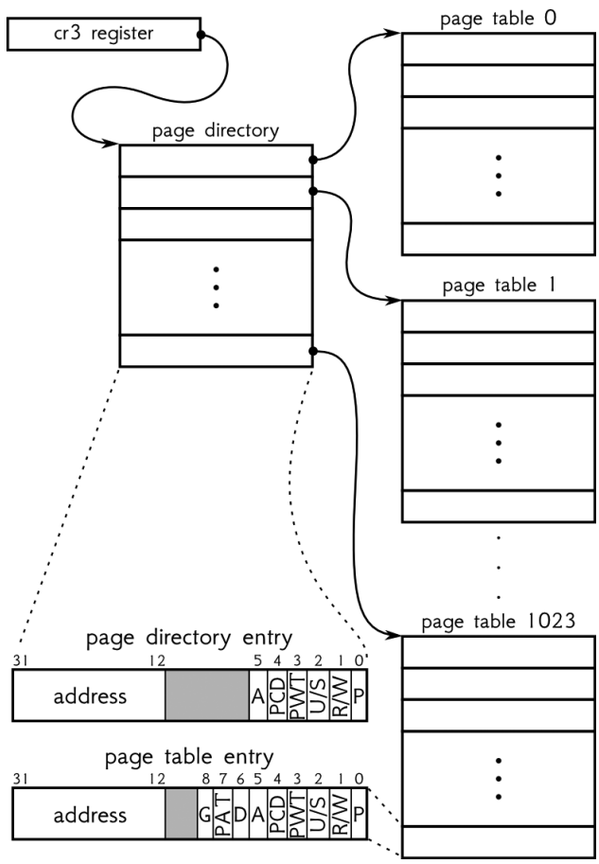
```
#define PAGE_TO_ADDR(pg) ((pg) * 4096)
#define BM_IDX(pg) ((pg) / 32)
#define BM_BIT(pg) ((pg) & 0x1f)
#define IS_FREE(pg) \
```

```
    ((bitmap[BM_IDX(pg)] & (1 << BM_BIT(pg))) == 0)

static void mark_page(int pg, int used)
{
    int idx = BM_IDX(pg);
    int bit = BM_BIT(pg);
    if(used) {
        bitmap[idx] |= 1 << bit;
    } else {
        bitmap[idx] &= ~(1 << bit);
    }
}

uint32_t alloc_phys_page(void)
{
    int i, idx, max, intr_state = get_intr_state();
    disable_intr();

    idx = last_alloc_idx;
    max = bmsize / 4;
    while(idx <= max) {
        /* if at least one bit is 0 then we have
         * a free page. find and allocate it. */
        if(bitmap[idx] != 0xffffffff) {
            for(i=0; i<32; i++) {
                int pg = idx * 32 + i;
                if(IS_FREE(pg)) {
                    mark_page(pg, USED);
                    last_alloc_idx = idx;

                    set_intr_state(intr_state);
                    return PAGE_TO_ADDR(pg);
                }
            }
        }
        idx++;
    }
    set_intr_state(intr_state);
    return 0;
}
```

Paging/virtual memory

Paging ονομάζουμε τη δυνατότητα του επεξεργαστή να μεταφράζει τις διευθύνσεις που χρησιμοποιούνται σε προσβάσεις της μνήμης πριν από κάθε πρόσβαση (ανάγνωση ή εγγραφή). Αυτό μας επιτρέπει να διαχειριστούμε τη μνήμη του συστήματος με μεγαλύτερη ευελιξία, να παρουσιάσουμε ένα απλό μοντέλο μνήμης στα processes όπου καθένα έχει στη διάθεση του ξεχωριστό, συνεχόμενο **virtual address space**, μεγέθους 4GB. Επίσης, μας επιτρέπει να επαναχρησιμοποιήσουμε εύκολα όποια τμήματα της μνήμης δεν χρησιμοποιούνται συχνά, αποθηκεύοντας τα περιεχόμενά τους κάπου, ώστε να τα επαναφέρουμε αργότερα, αν χρειαστεί **(swapping)**.

Με το paging ενεργοποιημένο, η μνήμη χωρίζεται σε pages των 4Kb και κάθε page φυσικής μνήμης μπορεί να γίνει map σε οποιοδήποτε virtual page θέλουμε, δηλαδή, σε οποιαδήποτε 4KB-aligned virtual διεύθυνση μέσα στα 4GB του address space. Αυτό γίνεται μέσω ενός πίνακα που αντιστοιχίζει φυσικές διευθύνσεις σε εικονικές διευθύνσεις (ανά page), που ονομάζεται **page table**. Αργότερα, όταν θα υλοποιήσουμε processes, θα δούμε ότι κάθε process μπορεί να



1 **Page tables.**

έχει το δικό του page table, το δικό του mapping, δηλαδή, μεταξύ virtual και φυσικών διευθύνσεων.

Ο x86 χρησιμοποιεί page tables δύο επιπέδων για τη μετάφραση των virtual διευθύνσεων σε φυσικές διευθύνσεις. Κατ' αρχάς, ο **cr3 register** δείχνει σε έναν πίνακα, που ονομάζεται **page directory.**

Το page directory περιέχει 1.024 διευθύνσεις φυσικής μνήμης (επί 4 bytes κάθε πεδίο συνολικά, πιάνει 4.096 bytes, ακριβώς ένα page), που δείχνουν σε ένα page table ίδιου μεγέθους, το οποίο με τη σειρά του περιέχει τις φυσικές διευθύνσεις όπου γίνονται map 1.024 virtual pages (βλ. εικόνα 1). Ακριβέστερα, το page table και το page directory περιέχουν πεδία των οποίων τα άνω 20 bits είναι τα αντίστοιχα bits της διεύθυνσης, ενώ τα υπόλοιπα 12 που σε μία page-aligned διεύθυνση θα ήταν μηδενικά, χρησιμοποιούνται για **attribute bits**, όπως φαίνεται στο σχήμα 1.

Τα ανώτερα 20 bits μίας 32bit διεύθυνσης μπορούμε να τα θεωρήσουμε ως το νούμερο του page και τα κατώτερα 12 bits offset μέσα σε αυτό το page (2^12 = 4.096). Οταν ο επεξεργαστής χρειαστεί να μεταφράσει μία διεύθυνση, πρέπει να βρει το page table entry που λέει σε ποια physical διεύθυνση αντιστοιχεί.

Για να το κάνει αυτό, παίρνει τα πρώτα 10 bits του page number και τα χρησιμοποιεί ως index στο page directory για να εντοπίσει το page table (2^10 = 1.024). Κατόπιν, τα επόμενα 10 bits μπορούν να χρησιμοποιηθούν ως index σε αυτό το page table, για να ανακτηθεί η φυσική διεύθυνση και τα διάφορα attributes του mapping. Αν τα attribute bits του

page directory ή page table entry που διαβάζονται στην παραπάνω διαδικασία έχουν στο πρώτο (present) bit 0, το page θεωρείται ότι δεν υπάρχει στη μνήμη και «σηκώνεται» page fault.

Η συνάρτηση **virt_to_phys** υλοποιεί την παραπάνω διαδικασία, για να μπορούμε να κάνουμε μετατροπή από virtual σε physical διευθύνσεις όποτε χρειαστεί.

```
#define PGENT_ADDR_MASK 0xfffff000
#define ADDR_TO_PAGE(x) ((x) >> 12)
#define ADDR_TO_PGTBL(x) ((x) >> 22)
#define ADDR_TO_PGTBL_PG(x) (((x) >> 12) & 0x3ff)
#define ADDR_TO_PGOFFS(x) ((x) & 0xfff)

uint32_t virt_to_phys(uint32_t vaddr)
{
    uint32_t pgaddr, *pgtbl;
    int diridx = ADDR_TO_PGTBL(vaddr);
    int pgidx = ADDR_TO_PGTBL_PG(vaddr);

    if(!(pgdir[diridx] & PG_PRESENT)) {
        panic("page table not present\n");
    }
    pgtbl = (uint32_t*)(pgdir[diridx] &
            PGENT_ADDR_MASK);
    if(!(pgtbl[pgidx] & PG_PRESENT)) {
        panic("page not present\n");
    }
    pgaddr = pgtbl[pgidx] & PGENT_ADDR_MASK;
    return pgaddr | ADDR_TO_PGOFFS(vaddr);
}
```

Οπως φαίνεται από τα παραπάνω, για κάθε πρόσβαση στη μνήμη ο επεξεργαστής πρέπει να κάνει άλλες δύο αναγνώσεις μόνο και μόνο για να κάνει τη μετάφραση της διεύθυνσης. Κάτι τέτοιο, φυσικά, είναι πολύ αργό, γι' αυτό υπάρχει μία cache από πρόσφατες μεταφράσεις που αντιστοιχίζει virtual σε physical pages, που λέγεται **translation lookaside buffer (TLB)**. Αυτή η λεπτομέρεια είναι σημαντική, γιατί είναι δική μας ευθύνη να ακυρώσουμε την cached μετάφραση στο TLB, όταν αλλάζουμε το αντίστοιχο mapping στο page table. Αυτό γίνεται για ένα συγκεκριμένο page, εκτελώντας την εντολή **invlpg**, κάτι που κάνει η συνάρτηση **fluhs_tlb_addr**, ή για όλες τις cached μεταφράσεις, γράφοντας στον cr3 register, κάτι που κάνει η **flush_tlb (vm-asm.S)**.

Οταν ξεκινά ο επεξεργαστής, το paging είναι απενεργοποιημένο και όλα τα accesses στη μνήμη γίνονται απευθείας με φυσικές διευθύνσεις Ενεργοποιούμε το paging, θέτοντας το bit 31 του cr0, ενώ μπορούμε να το απενεργοποιήσουμε ξανά, καθαρίζοντάς το. Αυτό κάνουν οι συναρτήσεις **enable_paging** και **disable_paging** στο **vm-asm.S**. Εάν, όμως, ενεργοποιήσουμε το paging, χωρίς να έχουμε δώσει στον επεξεργαστή σωστά page tables, θα δημιουργηθεί τριπλο-fault και reset, γιατί δεν θα μπορεί να βρει την επόμενη προς εκτέλεση εντολή. Το πρώτο πράγμα που κάνει, λοιπόν, η **init_vm** στο **vm.c**, είναι να είναι να καλέσει την **alloc_phys_page**, για να δεσμεύσει ένα page μνήμης, το οποίο θα χρησιμοποιηθεί ως **page directory**, και καλεί την **set_pgdir_addr**, που θέτει στον cr3 τη διεύθυνση αυτού του page.

Το επόμενο βήμα είναι να φροντίσουμε ώστε το mapping των pages που καταλαμβάνει ο κώδικας και τα δεδομένα του πυρήνα να είναι προσπελάσιμα στις ίδιες διευθύνσεις ασχέτως αν είναι ενεργοποιημένο η απενεργοποιημένο το paging, έτσι ώστε να μην προκύψουν προβλήματα κατά την αλλαγή. Αυτό το πετυχαίνουμε, καλώντας τη **map_mem_range**, με ίδια αρχική virtual και physical διεύθυνση για όλο το εύρος των διευθύνσεων που ανήκουν στο kernel image, κάτι που το βρίσκουμε, καλώντας την **get_kernel_mem_range** που ορίζεται στο mem.c. Η map_mem_range με τη σειρά της καλεί τη **map_page_range**, η οποία loοράρει για όλα τα pages στο διάστημα που της ζητήθηκε και καλεί τη map_page για καθένα από αυτά.

Η **map_page** και η αντίστροφή της, **unmap_page**, είναι οι μόνες συναρτήσεις που, τελικά, κάνουν allocate μνήμη για page tables και γράφουν σε αυτά, για να δημιουργήσουν ή να καταργήσουν mappings. Η διαδικασία είναι απλή και παρόμοια με τον αλγόριθμο της **virt_to_phys** που είδαμε παραπάνω. Πρώτα βρίσκουμε το σωστό page directory entry με τα άνω 10 bits του 20 bit page number και κοιτάμε αν όντως δείχνει σε valid page table, ελέγχοντας το present bit. Εάν δεν υπάρχει το page table, καλούμε την **alloc_phys_page** για να δεσμεύσουμε μνήμη γι' αυτό, τοποθετούμε τη διεύθυνσή του στο page directory entry, και θέτουμε το present bit σε 1. Κατόπιν, πηγαίνουμε στο page table και βρίσκουμε το entry που αντιστοιχεί στο page που θέλουμε να κάνουμε map, χρησιμοποιώντας τα επόμενα 10 bits ως index στο page table, όπου γράφουμε τη φυσική διεύθυνση του page και ό,τι attribute bits μάς ζητηθεί, καθώς, βεβαίως και το present bit.

Μία τελευταία λεπτομέρεια που πρέπει να αναφέρουμε όσον αφορά στο map/unmap, είναι ότι εφόσον ενεργοποιήσουμε το paging, όλες οι προσβάσεις της μνήμης γίνονται με virtual διευθύνσεις. Αυτό μας δυσκολεύει ελαφρώς, γιατί δεν μπορούμε να ακολουθήσουμε απλώς τις διευθύνσεις που είναι γραμμένες στον cr3 και στο page directory για να βρούμε και να αλλάξουμε το page table, μια και αυτές είναι φυσικές διευθύνσεις. Θα πρέπει να είναι τα ίδια τα page tables mapped κάπου για να μπορούμε να τα προσπελάσουμε. Μία απλή λύση θα ήταν να απενεργοποιήσουμε το paging, μπαίνοντας στην map_page, να δουλέψουμε με physical addresses απευθείας και να ενεργοποιήσουμε ξανά το paging μόλις τελειώσουμε. Το ίδιο ισχύει και για τις unmap_page και virt_to_phys, που, όπως δείξαμε προηγουμένως, θεωρεί ότι δουλεύει σε φυσική μνήμη.

Recursive page tables

Μία καλύτερη λύση που δεν απαιτεί να ανοιγοκλείνουμε συνεχώς το paging, είναι να εκμεταλλευτούμε την ικανότητα του x86 να ακολουθεί recursive page tables, ώστε να έχουμε μονίμως mapped όλα τα page tables και το page directory στα τελευταία 4MB του virtual address space **(ffc00000 - ffffffff)**. Το βρόμικο trick που θα χρησιμοποιήσουμε, είναι να βάλουμε στο τελευταίο entry του page directory τη διεύθυνση του ίδιου του page directory.

Τι πετυχαίνουμε πρακτικά με αυτό; Ας πούμε ότι διαβάζουμε από τη διεύθυνση **ffc00000**. Τα πρώτα 10 bits που μας δίνουν το page directory entry, είναι η τιμή 1.023. Δηλαδή, πάμε να βρούμε το page table που θα μας πει πού είναι το page που ψάχνουμε, στη διεύθυνση που περιέχεται στο τελευταίο από τα 1.024 entries του page directory, το οποίο, όπως είπαμε, περιέχει τη διεύθυνση του ίδιου του page directory. Τα επόμενα 10 bits είναι 0, άρα θα κοιτάξουμε στο πρώτο entry του page table που δεν είναι άλλο από το page directory, για να βρούμε το page που ψάχνουμε. Αλλά τι περιέχεται εκεί; Φυσικά, η διεύθυνση του πρώτου page table. Αρα, τελικά, η τιμή που διαβάζουμε ή γράφουμε στη διεύ-

θυνση ffc00000, είναι το πρώτο entry του πρώτου page table, κάτι που σημαίνει ότι μπορούμε να το τροποποιήσουμε για να κάνουμε map/unmap σελίδες χωρίς κανένα πρόβλημα. Διαβάστε ξανά την παραπάνω παράγραφο, μέχρι να σας συνεπάρει η ομορφιά και η απλότητα αυτής της λύσης.

Από τα παραπάνω συμπεραίνουμε ότι αν θέλουμε να διαβάσουμε ή να τροποποιήσουμε κάποιο entry του page directory, αυτό θα το βρούμε mapped στα τελευταία 4KB του address space **(fffff000 - ffffffff)**.

Συνοπτικά, η μετάφραση της διεύθυνσης fffff000: Τα πρώτα 10 bits είναι 1.023 άρα πάμε στο τελευταίο entry του page directory, το οποίο δείχνει στο page directory. Τα επόμενα 10 bits είναι επίσης 1.023, οπότε πάμε στο τελευταίο entry του page table που είναι το page directory, για να βρούμε το page, όπου, φυσικά, και πάλι βρίσκουμε τη διεύθυνση του page directory.

Ολα αυτά είναι υλοποιημένα στην init_vm στο vm.c ως εξής:

```
pgdir[1023] = ((uint32_t)pgdir & 0xfffff000) |
PG_PRESENT;
pgdir = (uint32_t*)0xfffff000;
```

Επίσης, το macro **PGTBL** μάς δίνει virtual memory pointer στο page table που θα του ζητήσουμε με το νούμερό του.

```
#define PGTBL(x) ((uint32_t*)(0xffc00000+4096*(x)))
```

Ολες οι συναρτήσεις που χρειάζονται πρόσβαση στα page tables, χρησιμοποιούν πλέον αυτό το macro για να τα κάνουν access. Για παράδειγμα, δείτε στο συνοδευτικό κώδικα τις αλλαγές που έχουν γίνει στη **virt_to_phys** που είδαμε προηγουμένως, για να δουλέψει σωστά και με το paging ενεργοποιημένο.

Διαχείριση εικονικής μνήμης

Μία απόφαση που θα μας διευκολύνει αργότερα, όταν υλοποιήσουμε processes και system calls, είναι να κρατάμε πάντα mapped τη μνήμη του kernel στο address space όλων τον processes. Ετσι, π.χ., αν χρειαστεί να αντιγράψουμε ένα κομμάτι μνήμης από I/O buffer του πυρήνα σε κάποιο buffer του process που κάλεσε το read system call, μπορούμε να το κάνουμε με μία απλή **memcpy**, αφού θα είναι ήδη mapped και τα δύο.

Ενας καλός τρόπος να διαχειριστούμε το virtual address space, κρατώντας υπόψη τα παραπάνω, και την απαίτηση να μπορούμε εύκολα να δεσμεύουμε συνεχόμενα διαστήματα από virtual pages, είναι να χρησιμοποιήσουμε δύο **linked lists** από ελεύθερα page ranges, μία για το user κομμάτι του address space **(0 – bfffffff)** και μία για το kernel κομμάτι του address space **(c0000000 – ffffffff).**

Η **pgalloc** και η **pgfree**, στο **vm.c** διαχειρίζονται αυτές τις δύο λίστες και μοιράζουν ή ελευθερώνουν μνήμη. Η pgalloc παίρνει ως παράμετρο πόσα pages θέλουμε και αν τα θέλουμε για kernel ή user χρήση, ώστε να κοιτάξει στην κατάλληλη λίστα. Οταν βρει ένα ελεύθερο range από pages που να χωρά το πλήθος pages που ζητήσαμε, κατ' αρχάς, το αφαιρεί από τη λίστα με τα ελεύθερα διαστήματα. Κατόπιν, καλεί τη map**_page_range** με virtual address την αρχή του διαστήματος που έκανε allocate και physical address -1, ώστε να γίνει το mapping σε physical pages που θα γίνουν allocate, καλώντας την **alloc_phys_page**. Να σημειωθεί ότι το αν τα pages φυσικής μνήμης που θα χρησιμοποιηθούν είναι συνεχόμενα ή όχι, δεν έχει καμία απολύτως σημασία, μια και αφού θα γίνουν map σε συνεχόμενα virtual pages, εμείς τα βλέπουμε σαν συνεχόμενο κομμάτι μνήμης. Οταν κληθεί

2 **Debugging output του πυρήνα.**

η pgfree, προστίθεται το εύρος των virtual διευθύνσεων στην κατάλληλη λίστα και, αν είναι δυνατόν, γίνεται σύμπτυξη της λίστας, καλώντας τη συνάρτηση **coalesce**, ώστε αν το range που ελευθερώσαμε, είναι γειτονικό με κάποιο υπάρχον free range, να μεγαλώσει το υπάρχον range, αντί να κολλήσουμε άλλο ένα node στη λίστα. Επίσης, για καθένα από τα pages καλούμε τη **virt_to_phys**, για να δούμε ποιο physical page είναι mapped εκεί και να το ελευθερώσουμε με τη **free_phys_page.**

Η **init_vm** κάνει initialize τον **virtual page allocator**, χρησιμοποιώντας ένα στατικό **page_range struct** για το αρχικό kernel free range list. Από εκεί και εμπρός, όταν χρειαζόμαστε επιπλέον nodes για τη λίστα, μπορεί να χρησιμοποιηθεί η **alloc_node**, η οποία κατ' αρχάς κοιτά αν υπάρχουν ελεύθερα nodes σε ένα pool με αχρησιμοποίητα nodes και επιστρέφει ένα από αυτά. Αν δεν έχουμε nodes στο pool, κάνουμε allocate ένα page με την **pgalloc**, το σπάμε σε 256 **page_range nodes**, τα κάνουμε link μεταξύ τους και τα βάζουμε στη λίστα, από την οποία, τελικά, παίρνουμε το πρώτο και το επιστρέφουμε.

Δοκιμάζοντας το VM

Διάφορα σημεία του memory management κώδικα αυτή τη στιγμή βγάζουν debugging output για να μας πουν τι ακριβώς κάνουν. Οπότε, αν bootαρουμε τον υπολογιστή ή τον emulator με τον πυρήνα που συνοδεύει το άρθρο, θα δούμε να τυπώνονται διάφορες σχετικές πληροφορίες, όπως στην **εικόνα 2**, που είναι φωτογραφία από την οθόνη μου.

Ασκήσεις για τον αναγνώστη

Δοκιμάστε να προσθέσετε μία υλοποίηση των standard C συναρτήσεων malloc και free στην klibc, ώστε να μπορεί ο πυρήνας να κάνει εύκολα allocate κομμάτια μνήμης οτιδήποτε μεγέθους και όχι απαραίτητα πολλαπλάσια του page size. Η υλοποίηση αυτών των συναρτήσεων είναι απλή υπόθεση, χρησιμοποιώντας τις pgalloc και pgfree που είδαμε παραπάνω. Στείλτε μου τις απαντήσεις σας, για να αδράξετε αιώνια υστεροφημία μέσα από τις σελίδες του περιοδικού, αλλιώς θα βρείτε στον κώδικα του επόμενου τεύχους τη δική μου υλοποίηση.