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

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

Εισαγωγή στο kernel development

Το πρώτο από δύο άρθρα που θα περιγράψουν την υλοποίηση processes, multitasking, system calls και scheduling στον πυρήνα.

Για Smartphones

Εργαλεία: GCC, GNU make, GRUB, qemu

Δυσκολία: ★★★★★

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

Το καλοκαίρι πέρασε και ήρθε ο καιρός να βουτήξουμε ξανά στον κώδικα του πυρήνα και να γράψουμε ένα από τα πιο βασικά και πιο πολύπλοκα κομμάτια του μέχρι στιγμής. Ήρθε η ώρα να υλοποιήσουμε processes! Το αντικείμενο είναι κάπως μεγάλο και θα χρειαστούμε δύο άρθρα για να το καλύψουμε, οπότε ας μη χάνουμε χρόνο: Κατεβάστε το συνοδευτικό κώδικα του άρθρου και ας ξεκινήσουμε αμέσως.

Γενικά περί διεργασιών

Σε ένα τυπικό λειτουργικό σύστημα, τα προγράμματα των χρηστών που εκτελούνται, απαρτίζονται από ένα η περισσότερα processes (διεργασίες). Δουλειά του πυρήνα είναι να παρέχει υπηρεσίες σε αυτά τα processes, να διαχειρίζεται το hardware εκ μέρους τους και να διαχειρίζεται το χρόνο εκτέλεσης που διατίθεται σε καθένα από αυτά, εναλλάσσοντας πολύ γρήγορα ποιο εκτελείται κάθε φορά, ώστε να δίνεται η ψευδαίσθηση ότι εκτελούνται ταυτόχρονα. Και όλα αυτά ενώ διατηρεί ένα απλό μοντέλο εκτέλεσης, όπου κάθε process θεωρεί ότι του διατίθεται ο υπολογιστής κατ' αποκλειστικότητα. Για να μπορέσει να παρουσιάσει αυτό το απλό μοντέλο εκτέλεσης στα processes, ο πυρήνας πρέπει να κάνει δύο βασικά πράγματα. Κατ' αρχάς, πρέπει να αποθηκεύει στη μνήμη την κατάσταση του επεξεργαστή όταν αποφασίζει να διακόψει την εκτέλεση του ενεργού process, ώστε να μπορεί να τον επαναφέρει ακριβώς στην ίδια κατάσταση, όταν έρθει

> **Ο πυρήνας δεν εκτελείται στο παρασκήνιο παράλληλα με τα προγράμματα των χρηστών, αντίθετα περνά την εκτέλεση σε κάποιο user process.**

η ώρα να συνεχίσει από εκεί όπου σταμάτησε. Κατά δεύτερον, πρέπει να παρέχει ξεχωριστό address space σε κάθε process, ώστε όταν ένα process γράφει κάτι στη μνήμη, να μην επηρεάζει τη μνήμη των άλλων.

Το πιο σημαντικό που πρέπει να κατανοήσουμε για να γίνει ξεκάθαρο το πώς λειτουργεί ο πυρήνας, πώς παρέχει υπηρεσίες στα processes και πώς τα εναλλάσσει, είναι το εξής: Ο πυρήνας δεν εκτελείται στο παρασκήνιο παράλληλα με τα προγράμματα των χρηστών, αντίθετα περνά την εκτέλεση σε κάποιο user process και δεν εκτελείται ξανά, μέχρι εκείνο να του δώσει πίσω την σκυτάλη, ζητώντας κάποια υπηρεσία, σηκώνοντας software interrupt ή εάν πρέπει να χειριστεί κάποιο hardware interrupt.

Αν, όμως, ο kernel δεν εκτελείται παρά μόνο όταν του ζητηθεί, πώς μπορεί να διακόψει την εκτέλεση ενός process βιαίως για να δώσει χρόνο σε κάποιο άλλο; Μα, φυσικά, χρησιμοποιώντας τον timer του υπολογιστή. Στο προηγούμενο άρθρο βάλαμε τον timer να σηκώνει interrupts σε τακτά διαστήματα (250 φορές το δευτερόλεπτο). Όταν συμβαίνει αυτό, παίρνει τον έλεγχο ο πυρήνας και αυτό που τον βάλαμε να κάνει μέχρι στιγμής, είναι να αυξάνει απλώς μία μεταβλητή (ticks) κατά ένα. Από εδώ και στο εξής θα χρησιμοποιήσουμε αυτό το timer interrupt για να ελέγχουμε ανά τακτά διαστήματα αν το ενεργό process ξεπέρασε το χρόνο που θέλουμε να του διαθέσουμε, οπότε το σταματάμε για να δώσουμε χρόνο εκτέλεσης σε κάποιο άλλο.

Privilege Levels

Μέχρι στιγμής, όλος ο κώδικας που γράψαμε, εκτελείται σε privilege level 0, δηλαδή, σε kernel mode, κάτι που πρακτικά σημαίνει ότι έχει πρόσβαση σε όλη τη μνήμη και μπορεί να διαχειριστεί όλο το hardware κατά βούληση. Αυτό είναι προφανώς ανεπιθύμητο για τα user processes, γι' αυτό θέλουμε να φροντίσουμε να εκτελούνται στο περιορισμένο level 3 του επεξεργαστή. Σε αυτό το επίπεδο απαγορεύεται η εκτέλεση εντολών που μπορεί να επηρεάσουν δομές που διαχειρίζεται ο πυρήνας, όπως page tables, segment descriptors, interrupt vectors κ.λπ., όπως και οι I/O εντολές τις οποίες χρησιμοποιούμε για να χειριστούμε τις συσκευές του συστήματος.

Για να το πετύχουμε αυτό, κατ' αρχάς πρέπει να τροποποιήσουμε τον πίνακα GDT (Global Descriptor Table), προσθέτοντας δύο καινούργια segment descriptors για το user code segment και το user data segment. Η ουσιαστική διαφορά μεταξύ αυτών και των kernel code και data segments που έχουμε ήδη, είναι ότι βάζουμε στο πεδίο dpl (descriptor privilege level) την τιμή 3 αντί για 0 (βλ. init_segm στο segm.c). Οι περιορισμοί που αναφέραμε παραπάνω, εφαρμόζονται από τον επεξεργαστή όταν εκτελείται κώδικας με τον cs register να περιέχει selector για segment με dpl 3. Αντίστοιχα, όταν γίνει πρόσβαση στη μνήμη μέσω data segment dpl 3, ο επεξεργαστής επιτρέπει την πρόσβαση μόνο σε pages, στο page table entry των οποίων έχουμε θέσει το user bit. Φυσικά, ο κώδικας που εκτελείται σε user mode, δεν επιτρέπεται να αλλάξει τους segment selectors του, αλλιώς όλα τα παραπάνω δεν θα είχαν ιδιαίτερο νόημα.

Είσοδος και έξοδος από kernel mode

Η αλλαγή του privilege level από 3 σε 0 (user->kernel) γίνεται αυτόματα όταν σηκωθεί interrupt ενώ είμαστε σε user mode. Η διαδικασία εισόδου στον interrupt handler διαφέρει ελαφρώς αν αυτό γίνει από user mode αντί για kernel mode που είχαμε δει μέχρι τώρα. Η βασική διαφορά είναι ότι κατά την είσοδο στον interrupt handler ο επεξεργαστής αλλάζει αυτόματα τους registers ss (stack segment) και esp (stack pointer), ώστε να χρησιμοποιηθεί από τον kernel διαφορετικό stack από αυτό που χρησιμοποιούσε ο user κώδικας. Στο

καινούργιο stack γίνονται push επιπλέον οι παλιές τιμές αυτών των δύο registers, πέρα από όλα τα υπόλοιπα που είδαμε στο δεύτερο άρθρο **(σχήμα 1)**.

Το interrupt frame περιέχει την προηγούμενη τιμή του cs, και, φυσικά, και του eip, τα οποία επαναφέρει ο επεξεργαστής όταν γίνει επιστροφή από interrupt (εντολή iret). Αν, λοιπόν, αυτή η αποθηκευμένη τιμή του cs έχει dpl 3, τότε όταν ο επεξεργαστής την επαναφέρει στον cs, γίνεται ξανά μετάβαση σε user mode και γυρνάμε αυτόματα στο stack του user process (επαναφέρονται οι αποθηκευμένες τιμές των ss και esp). Τις τιμές που θα θέσει στους ss και esp ο επεξεργαστής κατά την είσοδο σε kernel mode, τις παίρνει από ένα ειδικό segment, τον descriptor του οποίου πρέπει, φυσικά, να τοποθετήσουμε στον GDT, το Task State Segment (TSS). Δεν θα μπούμε σε λεπτομέρειες για τις πιθανές χρήσεις του TSS, αλλά περιληπτικά είναι μία δομή με συγκεκριμένη μορφή, όπως φαίνεται στο struct task_state στο αρχείο tss.h, στην οποία περιμένει να βρει ο επεξεργαστής τη διεύθυνση και τον selector για το stack που θα χρησιμοποιηθεί κατά τη μετάβαση σε οποιοδήποτε privilege level μικρότερο του 3. Εμείς αγνοούμε τα stacks για privilege levels 1 και 2, μια και δεν μας είναι χρήσιμα, καθώς και όλα τα υπόλοιπα περιεχόμενα της συγκεκριμένης δομής.

Στον κώδικα του interrupt entry (intr-asm.S), οι αλλαγές που χρειάστηκαν, είναι ελάχιστες και δεν έχει νόημα να τον επαναλάβουμε εδώ. Απλώς, μπαίνοντας πλέον στο interrupt, κάνουμε push όλους τους selector registers, εκτός από τον cs και τον ss που γίνονται αυτόματα, και τους κάνουμε pop πάλι πριν από το iret.

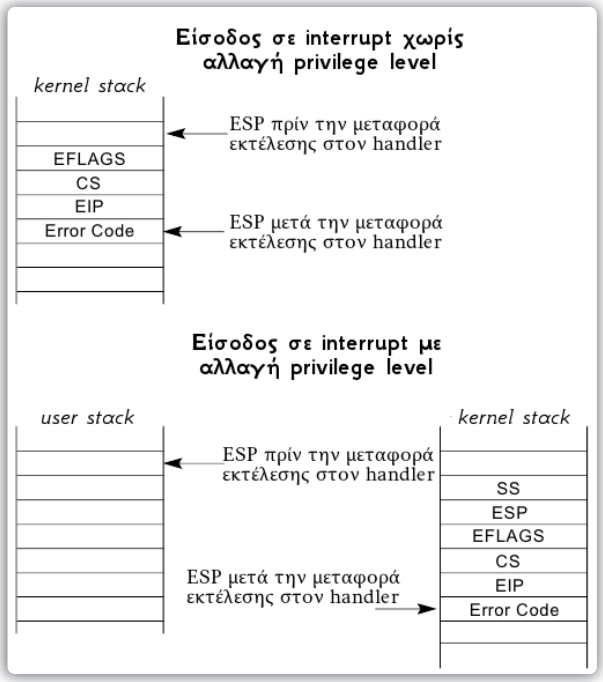
Ουσιαστικότερη είναι η συνάρτηση init_proc στο αρχείο proc.c που κάνει allocate χώρο για το TSS, θέτει σε αυτό τη σωστή τιμή για τον ss του kernel και καλεί τη set_tss (segm.c), για να μπει ο descriptor του στον GDT και να εκτελεστεί η εντολή ltr που δίνει στον επεξεργαστή τον selector για το TSS.

```
static struct task_state *tss;
void init_proc(void)
{
    /* allocate a page for the task state segment */
    int tss_page = pgalloc(1, MEM_KERNEL);
    tss = (void*)PAGE_TO_ADDR(tss_page);

    /* clear the tss and set the correct ss selector */
    memset(tss, 0, sizeof *tss);
    tss->ss0 = selector(SEGM_KDATA, 0);

    set_tss((uint32_t)tss);

    init_syscall();
    start_first_proc(); /* never returns */
}
```

Βλέπουμε ότι η init_proc κάνει επίσης initialize το μηχανισμό των system calls καλώντας την init_syscall και, τέλος, ξεκινά το πρώτο user process, καλώντας τη start_first_proc, την οποία θα δούμε παρακάτω.

System calls

Ο μηχανισμός που θα χρησιμοποιήσουμε για τα system calls είναι πολύ απλός. Όταν ένα process θέλει να ζητήσει κάτι από τον πυρήνα, τοποθετεί στον eax το νούμερο του system call, στους ebx, ecx, edx, esi, και edi τις παραμέτρους (όσες χρειάζονται για το κάθε system call) και σηκώνει



1 **Interrupt frame κατά την είσοδο από ίδιο ή διαφορετικό privilege level.**

το interrupt 128 εκτελώντας την εντολή int. Η εκτέλεση μεταφέρεται στον πυρήνα σε kernel mode και, όπως έχουμε πει σε παλαιότερο άρθρο, καταλήγει να εκτελεστεί η συνάρτηση που έχουμε θέσει για το συγκεκριμένο interrupt. Για το interrupt 128, λοιπόν, βάζουμε μία συνάρτηση, που χρησιμοποιεί την αποθηκευμένη τιμή του eax από το interrupt frame ως index σε έναν πίνακα από συναρτήσεις που υλοποιούν κάθε system call. Όταν επιστέψει η συνάρτηση του system call, παίρνουμε την τιμή που επέστρεψε και τη χρησιμοποιούμε για να αντικαταστήσουμε την αποθηκευμένη τιμή του eax στο stack frame, η οποία θα τοποθετηθεί στον eax register λίγο προτού επιστρέψουμε από το interrupt, με την popa που εκτελεί ο interrupt handler πριν από το iret.

Τα παραπάνω υλοποιούνται στο αρχείο syscall.c, ενώ οι συμβολικές τιμές που χρησιμοποιούνται για τους αριθμούς των system calls, ορίζονται στο header file syscall.h.

```
void init_syscall(void)
{
    sys_func[SYS_HELLO] = sys_hello;
    sys_func[SYS_SLEEP] = sys_sleep;
    sys_func[SYS_FORK] = sys_fork;
    sys_func[SYS_GETPID] = sys_getpid;

    interrupt(SYSCALL_INT, syscall);
}

static void syscall(int inum)
{
    struct intr_frame *frm = get_intr_frame();
    int idx = frm->regs.eax
    frm->regs.eax = sys_func[idx](frm->regs.ebx, frm-
>regs.ecx, frm->regs.edx, frm->regs.esi, frm->regs.edi);
}
```

Τα system calls που υλοποιήσαμε μέχρι στιγμής, είναι τα

τέσσερα που φαίνονται στο παραπάνω κομμάτι κώδικα. Όταν κληθεί το syscall hello, τυπώνεται ένα μήνυμα της μορφής «το process τάδε λέει hello», ώστε να βλέπουμε ποιο process τρέχει κάθε φορά. Η sleep παίρνει ως παράμετρο έναν αριθμό από δευτερόλεπτα και σταματά το ενεργό process για τουλάχιστον τόσο χρόνο, αφήνοντας άλλα processes να τρέξουν. Το fork θα το δούμε λεπτομερώς σε λίγο, μια και είναι η μέθοδος με την οποία δημιουργούνται άλλα processes. Τέλος, το getpid απλώς επιστρέφει το id του process που έκανε την κλήση.

Διαχείριση διεργασιών

Υπάρχουν διάφοροι τρόποι να γίνουν η διαχείριση, η δημιουργία και το scheduling της εκτέλεσης των processes. Εμείς θα ακολουθήσουμε μία οργάνωση βασισμένη χοντρικά στο μοντέλο του UNIX.

Σε αυτό το μοντέλο, ο μόνος τρόπος να δημιουργηθεί καινούργιο process, είναι μέσω της κλήσης fork που φτιάχνει ένα ακριβές αντίγραφο του process που την κάλεσε. Με αυτόν τον τρόπο φτιάχνεται μία σχέση γονέα-παιδιού μεταξύ των διεργασιών. Έτσι, δημιουργείται μία ιεραρχία από processes στο σύστημα που έχει ως ρίζα το πρώτο process (pid 1) το οποίο ξεκινά ο πυρήνας μετά το initialization του και ιστορικά λέγεται init.

Για κάθε process κρατάμε ένα structure με πληροφορίες (struct process στο proc.h), που περιέχει μεταξύ άλλων το id του, το id του parent, αν είναι ενεργό και μπορεί να εκτελεστεί ή περιμένει για κάτι (και τι είναι αυτό που περιμένει) κ.ά. Αυτά τα structures τα έχουμε σε έναν πίνακα και κάθε process απλώς χρησιμοποιεί τη θέση του πίνακα που αντιστοιχεί στο id του.

> **Μία σημαντική λεπτομέρεια όσον αφορά στην οργάνωση και στη λειτουργία των διεργασιών, είναι ότι κάθε process έχει το δικό του kernel stack.**

Μία σημαντική λεπτομέρεια όσον αφορά στην οργάνωση και στη λειτουργία των διεργασιών, είναι ότι κάθε process έχει το δικό του kernel stack. Αυτό μας επιτρέπει να σταματάμε την εκτέλεση κάποιου system call που χρειάζεται να περιμένει για κάτι και να μπορούμε να συνεχίσουμε αργότερα από το σημείο όπου σταματήσαμε, κάτι που απλοποιεί πολύ την υλοποίηση των system calls. Για παράδειγμα, η read, όταν διαβάζει από δίσκο –κάτι που θα υλοποιήσουμε σε μεταγενέστερο άρθρο– μπορεί απλώς να ξεκινήσει τη διαδικασία ανάγνωσης και να σταματήσει μέχρι να γίνουν διαθέσιμα τα δεδομένα από το δίσκο, οπότε να συνεχιστεί η εκτέλεση του κώδικα μέσα στον kernel που θα αντιγράψει τα δεδομένα στον buffer που έχει δώσει ο user και θα επιστρέψει σε user space. Με παρόμοιο τρόπο δουλεύει και η sleep που θα περιγράψουμε λεπτομερώς στο επόμενο άρθρο.

Το πρώτο process (init)

Εφόσον όλα τα processes δημιουργούνται ως αντίγραφα του γονέα τους μέσω της fork, το πρώτο process πρέπει να κατασκευαστεί χειροκίνητα από τον kernel. Αυτό το κάνει η συνάρτηση start_first_proc, στο proc.c. Κανονικά θα θέλαμε να φορτώσουμε το image του init process από το δίσκο, αλλά αφού δεν έχουμε ακόμη γράψει filesystem, πρέπει να αρκεστούμε σε κάποιου είδους πρόχειρο hack που θα μας επιτρέψει να δοκιμάσουμε τον κώδικα που γράφουμε για τα processes προτού αποκτήσουμε filesystem και executable loader. Η λύση στην οποία κατέληξα, είναι να γραφεί μία μικρή συνάρτηση σε assembly (test_proc.S) που θα γίνει compile μαζί με τον κώδικα του kernel. Σε αυτή τη συνάρτηση τοποθετούμε, εκτός από το label του entry point, και ένα globally visible label στο τέλος της. Έτσι, μπορούμε να υπολογίσουμε το μέγεθος του κώδικα με μία απλή αφαίρεση και να κάνουμε allocate ένα κομμάτι user μνήμης, να αντιγράψουμε τον κώδικα εκεί και να αρχίσει η εκτέλεση του πρώτου process από αυτό το σημείο.

```
/* allocate a chunk of memory for the process image */
proc_size_pg = (test_proc_end - test_proc) / PGSIZE + 1;
img_start_pg = pgalloc(proc_size_pg, MEM_USER);
img_start_addr = PAGE_TO_ADDR(img_start_pg);
memcpy((void*)img_start_addr, test_proc, proc_size_pg *
PGSIZE);
```

Έπειτα από αυτό, η start_first_proc γεμίζει το process structure και κάνει allocate kernel stack και user stack για το process. Καλεί την add_proc (sched.c) που προσθέτει στη λίστα ενεργών διεργασιών του scheduler το συγκεκριμένο process και καλεί τη set_current_pid, για να τεθεί η global μεταβλητή που μας λέει ποιο process εκτελείται ανά πάσα στιγμή. Επίσης, τοποθετεί τη διεύθυνση του kernel stack στο TSS, ώστε να χρησιμοποιηθεί στο επόμενο interrupt από user space.

Το πιο παράξενο κομμάτι είναι το πώς ξεκινά να εκτελείται το process. Αφού ο kernel, όπως είπαμε, εκτελείται πάντα ως απάντηση σε interrupt και επιστρέφει στο user space με επιστροφή από interrupt, η μόνη μέθοδος να ξεκινήσει να εκτελείται το process σε user mode, είναι να επιστρέψουμε σε αυτό από interrupt με την εντολή iret! Γι' αυτόν το λόγο, η start_first_proc κατασκευάζει ένα ψεύτικο interrupt frame με όλα τα στοιχεία που θέλουμε να μπουν στους διάφορους registers κατά την επιστροφή από interrupt και το περνά ως παράμετρο στη συνάρτηση intr_ret.

```
#define FLAGS_INTR_BIT (1 << 9)

struct intr_frame ifrm;
memset(&ifrm, 0, sizeof ifrm);
/* ss:esp after the priviledge switch */
ifrm.esp = PAGE_TO_ADDR(stack_pg) + PGSIZE;
ifrm.ss = selector(SEGM_UDATA, 3);
/* instruction pointer at the beginning of the image */
ifrm.eip = img_start_addr;
ifrm.cs = selector(SEGM_UCODE, 3);
/* make sure the user will run with interrupts enabled */
ifrm.eflags = FLAGS_INTR_BIT;
/* user data selectors should all be the same */
ifrm.ds = ifrm.es = ifrm.fs = ifrm.gs = ifrm.ss;

/* execute an iret with this stack frame */
intr_ret(ifrm);
```

Η συνάρτηση intr_ret, αφού παίρνει ως παράμετρο το ψεύτικο stack frame structure, αυτό γίνεται push στο stack προτού κληθεί η συνάρτηση. Αυτό σημαίνει ότι αν εξαιρέσουμε το return address που επίσης γίνεται push αυτόματα από την εντολή call, όταν μπαίνουμε στην intr_ret έχουμε ακριβώς τα δεδομένα στο stack που θα είχαμε αν ήμασταν στην intr_entry_common, έτοιμοι να ξεκινήσουμε να κάνουμε pop πράγματα και iret. Οπότε, η intr_ret αρκεί να ξεφορτωθεί το

return address από το stack προσθέτοντας 4 στον esp, προτού κάνει jump στη μέση του interrupt handler σε ένα label που προσθέσαμε αμέσως μετά την επιστροφή από την dispatch_intr.

```
    .globl intr_ret
intr_ret:
    add $4, %esp
    jmp intr_ret_local
```

Context switching

Context switching, δηλαδή, αλλαγή του process που εκτελείται, γίνεται όταν κληθεί η συνάρτηση schedule (sched.c). Αυτή η συνάρτηση καλείται από τον handler του timer interrupt (timer_handler στο timer.c), αφού πρώτα μειωθεί το ticks_left στο process structure του ενεργού process. Επίσης, καλείται και από τη συνάρτηση wait που χρησιμοποιείται όταν κάποιο process θέλει να σταματήσει να εκτελείται περιμένοντας κάτι.

Η schedule διατηρεί μία λίστα με processes που είναι έτοιμα να εκτελεστούν (run-queue) και πάντα διαλέγει το πρώτο σε αυτή τη λίστα προς εκτέλεση. Αν το ticks_left του process που εκτελείται (το πρώτο στη λίστα) έχει φτάσει στο 0 και υπάρχει και άλλο process στην λίστα που περιμένει να εκτελεστεί, αφαιρεί το process από την αρχή της λίστας και το τοποθετεί στο πίσω μέρος.

Σε κάθε περίπτωση περνά, τελικά, το id του πρώτου της λίστας στη συνάρτηση context_switch (proc.c), η οποία αναλαμβάνει να ανταλλάξει το ενεργό process με το καινούργιο, εάν είναι διαφορετικά.

Αν δεν υπάρχει κανένα process στο run-queue, τότε καλείται η συνάρτηση idle_proc, η οποία απλώς κάνει halt τον επεξεργαστή σε ένα loop με ενεργοποιημένα τα interrupts όσο συνεχίζει να είναι άδειο το run-queue.

```
void context_switch(int pid)
{
    struct process *prev = proc + last_pid;
    struct process *new = proc + pid;
    if(last_pid != pid) {
        set_current_pid(new->id);
        /* switch to the new process' address space */
        set_pgdir_addr(new->ctx.pgtbl_paddr);
        /* set the new kernel stack in tss */
        tss->esp0 = PAGE_TO_ADDR(new->kern_stack_pg)
+ KERN_STACK_SIZE;

        /* push all registers onto the stack */
        push_regs();
        switch_stack(new->ctx.stack_ptr, &prev-
>ctx.stack_ptr);
        /* restore registers from the NEW STACK */
        pop_regs();
    } else {
        set_current_pid(new->id);
    }
}
```

Βλέπουμε ότι η context_switch αλλάζει page tables και φροντίζει να αλλάξει το kernel stack σε αυτό του καινούργιου process, καλώντας τη switch_stack (proc-asm.S). Επίσης, φροντίζει να σώσει την κατάσταση των registers στο παλαιό stack προτού το αλλάξει και να επαναφέρει τους registers που είχε σώσει την τελευταία φορά που έτρεξε το καινούργιο process, κάνοντας pop από το νέο stack.

```
free: 0 - 9f400 (652288 bytes)
hole: 9f400 - a0000 (3072 bytes)
hole: f0000 - 100000 (65536 bytes)
free: 100000 - 7ff4000 (133156864 bytes)
hole: 7ff4000 - 8000000 (12288 bytes)
marking pages up to 10ed0c (page: 270) inclusive as used
System real-time clock: Tue Aug 16 07:42:30 2011
copied init process at: 10f000
process 1 is forking
process 1 says hello!
process 1 getpid
process 1 will sleep for 2 seconds
sleep(2000)
process 2 says hello!
process 2 getpid
process 2 will sleep for 4 seconds
sleep(4000)
idle loop is running
timer going off!!!
process 1 says hello!
process 1 getpid
process 1 will sleep for 2 seconds
sleep(2000)
idle loop is running
```

2 **Τα δύο πρώτα test processes του kernel.**

fork

Η fork (proc.c), όπως είπαμε, δημιουργεί ένα καινούργιο process, κοινοποιώντας το process που την κάλεσε. Το πρώτο πράγμα που πρέπει να κάνει η fork, είναι να εντοπίσει μία ελεύθερη θέση στο process table και να αναθέσει το αντίστοιχο pid στο καινούργιο process.

Μετά πρέπει να κάνει allocate kernel stack για το νέο process. Σε αυτό το kernel stack βάζουμε κατ' αρχάς το υπάρχον interrupt frame, ώστε το καινούργιο process να συνεχίσει την εκτέλεση από το ίδιο σημείο όπου ήταν και ο γονέας, όταν επιστρέψει σε user space.

Όμως, φροντίζουμε να βάλουμε την τιμή 0 στον eax αυτού του interrupt frame, ώστε να επιστρέψουμε 0 από τη fork στο child process (στο γονέα επιστρέφουμε το pid του καινούργιου process).

> **Η schedule διατηρεί μία λίστα με processes που είναι έτοιμα να εκτελεστούν (run-queue) και πάντα διαλέγει το πρώτο σε αυτή τη λίστα προς εκτέλεση.**

Τέλος, πρέπει να δημιουργήσει για το καινούργιο process ακριβές αντίγραφο της μνήμης του parent, κάτι που αναλαμβάνει να κάνει η clone_vm στο vm.c. Αυτή τη στιγμή, η clone_vm δεν αντιγράφει τη μνήμη του γονέα, αλλά κάνει τα δύο processes να μοιράζονται την ίδια μνήμη, αντιγράφοντας μόνο τα page tables, καθώς επίσης αλλάζει και τα attributes στα page tables, ώστε να είναι read-only η κοινή μνήμη. Αυτό γιατί στο επόμενο άρθρο θα υλοποιήσουμε την αντιγραφή του VM με copy-on-write. Έτσι, αν τώρα στο test_proc μετά την κλήση στη fork προσθέσουμε και μία εντολή που γράφει στη μνήμη, π.χ., push %eax, θα δούμε ότι σηκώνεται αμέσως page fault.

Αποτέλεσμα

Εκτελώντας τον κώδικα σε αυτή τη φάση, βλέπουμε ότι εκτελείται κανονικά ο κώδικας του test_proc.S ως user process, γίνεται το fork και μετά συνεχώς παίρνουμε μηνύματα από τα δύο processes που καλούν hello, getpid, και sleep σε loop **(εικόνα 2).**

Στο επόμενο τεύχος θα ολοκληρώσουμε την clone_vm υλοποιώντας copy-on-write, θα εξηγήσουμε πώς δουλεύει η sleep, καθώς και οι συναρτήσεις wait και wakeup στον scheduler, και θα φτιάξουμε και μερικά ακόμη βασικά system calls, όπως wait και exit. Μέχρι τότε, happy hacking.

•