Λειτουργικά Συστήματα Δραστηριότητα 3

2024-2025

AM: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

Περιεχόμενα

Άσκηση 1:

Τροποποιήστε το αρχικό module που φτιάξαμε ώστε να τυπώνει ένα διαφορετικό μήνυμα. Ακολουθήστε την ίδια διαδικασία για την φόρτωση/εκφόρτωσή του.

Βήματα:

1. Στον φάκελο /root/hello-world, δημιουργούμε ένα αρχείο hello.c. Γράφουμε τον κώδικα με παραλλαγμένα τα μηνύματα.

```
#include <linux/kernel.h>
```

```
#include <linux/module.h>
MODULE_DESCRIPTION("My hello kernel module");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");
static int my_init(void)
{
     printk("Hello world, from student\n");
return 0;
}
static void my_exit(void)
{
     printk("*******Goodbye friend*******\n");
}
module_init(my_init);
module_exit(my_exit);
```

2. Δημιουργούμε το κατάλληλο Makefile:

```
obj-m += hello.o
 PWD := $(CURDIR)

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3. Εκτελούμε τις παρακάτω εντολές στο terminal:

```
root@debian:~/hello-world# make  #κάνουμε μεταγλώττιση του module  στο αρχείο .c
make -C /lib/modules/6.1.0-26-amd64/build M=/root/hello-world modules
make[1]: Entering directory '/usr/src/linux-headers-6.1.0-26-amd64'
make[1]: Leaving directory '/usr/src/linux-headers-6.1.0-26-amd64'
root@debian:~/hello-world# modinfo hello.ko  #πληροφορίες για το αρχείο hello.ko
```

```
filename:       /root/hello-world/hello.ko
license:        GPL
author:         Me
description:    My hello kernel module
depends:
retpoline:      Y
name:           hello
vermagic:       6.1.0-26-amd64 SMP preempt mod_unload modversions
root@debian:~/hello-world# sudo insmod hello.ko  #φορτώνουμε το module στο σύστημα
root@debian:~/hello-world# sudo lsmod | grep hello  #ελέγχουμε αν φορτώθηκε
hello                  16384  0
root@debian:~/hello-world# sudo rmmod hello  #αφαιρούμε το module
```

4. Εκτελούμε αυτή την εντολή ώστε να ελέγξουμε τα logs του συστήματος:

```
root@debian:~/hello-world# journalctl --since "10 minutes ago" | grep kernel
Nov 19 06:43:41 debian kernel: Hello world, from student
Nov 19 06:44:13 debian kernel: *******Goodbye friend*******
Nov 19 06:44:16 debian kernel: No guest source window
```

Άσκηση 2:

Πηγαίνετε στον πηγαίο κώδικα του Linux kernel και διαβάστε τον ορισμό για το task_struct (στο αρχείο linux/include/linux/sched.h. Τί πληροφορίες αποθηκεύονται σε αυτή τη δομή; Τι αντιπροσωπεύει;

Το Linux kernel αποθηκεύει την λίστα των διεργασιών σε μια κυκλική διπλά συνδεδεμένη λίστα, που ονομάζεται **task list**. Κάθε στοιχείο αυτής της λίστας είναι τύπου **struct task_struct.** Το struct αυτό περιγράφει μια **διεργασία, όπως το proc στο xv6** και ορίζεται στο αρχείο **<linux/sched.h>**.

Η **task_struct** […] περιλαμβάνει όλες τις πληροφορίες που έχει και χρειάζεται το kernel για μια συγκεκριμένη διεργασία. Είναι μια σχετικά μεγάλη δομή δεδομένων, περίπου **1.7 kilobytes** σε ένα σύστημα 32-bit. Ωστόσο, αυτό το μέγεθος θεωρείται μικρό, δεδομένου ότι περιέχει όλες τις πληροφορίες για τη διεργασία. [από το βιβλίο Linux Kernel Development (Κεφάλαιο 3)]

Φυλάει κάποιες βασικές πληροφορίες όπως:

1. `pid`: Αναγνωριστικό διεργασίας (Process ID).
2. `tgid`: Αναγνωριστικό της ομάδας διεργασιών (Thread Group ID).
3. `state`: Τρέχουσα κατάσταση της διεργασίας.

Στα Linux υπάρχουν παραπάνω πιθανά states από ότι στο xv6. Όσον αφορά τα κύρια states που μπορεί να πάρει μια διεργασία, αρχικά υπάρχει η **1.TASK_RUNNING:** για είτε την τρέχουσα διεργασία είτε για μια δοεργασία έτοιμη να αρχίσει να εκτελείται, η οποία περιμένει δηλαδή στην ουρά.Ύστερα υπάρχει η **2.TASK_INTERRUPTIBLE** για μια διεργασία που κοιμάται (sleeping), αλλά μπορεί να ενεργοποιηθεί από συγκεκριμένα σήματα. Παραδείγματα αποτελούν οι διεργασίες που περιμένουν κάποια I/O ενέργεια. Στη συνέχεια υπάρχει η κατάσταση

3.TASK_UNINTERRUPTIBLE (Μη Διακοπτόμενη Αναμονή) για διεργασία που κοιμάται (sleeping) και δεν μπορεί να ξυπνήσει από σήματα. Χρησιμοποιείται όταν μια διεργασία βρίσκεται σε κρίσιμο τμήμα όπου η διακοπή δεν είναι ασφαλής, π.χ. διεργασία που αναμένει κάποια λειτουργία του hardware. Επιπλέον μια διεργασία μπορεί να είναι σε state **4.TASK_STOPPED** (Σταματημένη), δηλαδή η διεργασία να έχει σταματήσει από κάποιο σήμα. Η διεργασία θα παραμένει σε αυτήν την κατάσταση εκτός και αν επανεκκινηθεί μέσω κάποιου σήματος. Επιπλέον, υπάρχει το state **5.TASK_TRACED** (Υπό Εντοπισμό), για όταν μια διεργασία παρακολουθείται ή αποσφαλματώνεται. Η διεργασία παραμένει παγωμένη μέχρι να επιτραπεί η συνέχιση της εκτέλεσής της. Τέλος υπάρχει και το **6.TASK_DEAD** (Νεκρή) για όταν η διεργασία διαγράφεται και δεν είναι πλέον έγκυρη. Το kernel καθαρίζει τους πόρους της.

Υπάρχουν κάποια επιπλέον states όπως το **7.TASK_PARKED** χρησιμοποιείται για threads που έχουν ολοκληρώσει την εργασία τους και περιμένουν να επαναχρησιμοποιηθούν, ενώ το **8.TASK_WAKEKILL** και το **9.TASK_WAKING** σχετίζονται με την αφύπνιση διεργασιών, είτε λόγω fatal signal είτε κατά τη μετάβασή τους από την αναμονή στην εκτέλεση. Το **10.TASK_NOLOAD** χρησιμοποιείται για όποια διεργασία δεν επηρεάζει τη φόρτωση του συστήματος και δεν είναι κρίσιμη. Αντίθετα το **11.TASK_NEW** αφορά νεοδημιουργημένες διεργασίες που δεν έχουν προγραμματιστεί για εκτέλεση. Το **TASK_STATE_MAX** ορίζει την ανώτατη τιμή των καταστάσεων για έλεγχο σφαλμάτων (δεν είναι κατάσταση), και το **12.TASK_RTLOCK_WAIT** χρησιμοποιείται όταν μια διεργασία περιμένει αποκλειστικό real-time κλείδωμα. Στο πλαίσιο αναστολής λειτουργίας, το **13.TASK_FREEZABLE** και το **14.TASK_FROZEN** επιτρέπουν την προσωρινή παύση διεργασιών, ενώ το **15.__TASK_FREEZABLE_UNSAFE** αφορά πιο επικίνδυνες περιπτώσεις που παγώνουν, ανάλογα με τη διαμόρφωση του συστήματος. Αυτές οι καταστάσεις εξασφαλίζουν την ομαλή λειτουργία και διαχείριση ειδικών περιπτώσεων στο σύστημα.

4. `exit_state`: Κατάσταση κατά την έξοδο από τη διεργασία.

Πιθανές τιμές του exit_state:

```
 98  /* Used in tsk->__state: */
 99  #define TASK_RUNNING                    0x00000000
100  #define TASK_INTERRUPTIBLE              0x00000001
101  #define TASK_UNINTERRUPTIBLE            0x00000002
102  #define __TASK_STOPPED                  0x00000004
103  #define __TASK_TRACED                   0x00000008
104  /* Used in tsk->exit_state: */
105  #define EXIT_DEAD                       0x00000010
106  #define EXIT_ZOMBIE                     0x00000020
107  #define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
108  /* Used in tsk->__state again: */
109  #define TASK_PARKED                     0x00000040
110  #define TASK_DEAD                       0x00000080
111  #define TASK_WAKEKILL                   0x00000100
112  #define TASK_WAKING                     0x00000200
113  #define TASK_NOLOAD                     0x00000400
114  #define TASK_NEW                        0x00000800
115  #define TASK_RTLOCK_WAIT                0x00001000
116  #define TASK_FREEZABLE                  0x00002000
117  #define __TASK_FREEZABLE_UNSAFE        (0x00004000 * IS_ENABLED(CONFIG_LOCKDEP))
118  #define TASK_FROZEN                     0x00008000
119  #define TASK_STATE_MAX                  0x00010000
```

1.EXIT_ZOMBIE: Η διεργασία έχει τερματίσει την εκτέλεσή της και έχει μετατραπεί σε ζόμπι (zombie process), δηλαδή ο γονέας της περιμένει να ανακτήσει την κατάσταση εξόδου της (exit status) μέσω κλήσεων όπως wait() ή waitpid(). Σε αυτή την κατάσταση, η διεργασία δεν χρησιμοποιεί ενεργά πόρους (CPU, μνήμη), αλλά εξακολουθεί να υπάρχει στον πίνακα διεργασιών (process table).

2.EXIT_DEAD: Η διεργασία έχει ολοκληρωθεί πλήρως και αφαιρεθεί από τον πίνακα διεργασιών, αφού ο γονέας της διεργασίας ανέκτησε την κατάσταση εξόδου της. Η διεργασία αφαιρείται από τη λίστα των διεργασιών.

5. `sched_class`: Πληροφορίες για την κλάση του scheduler.
6. `prio` και `static_prio`: Προτεραιότητα της διεργασίας.
7. `cpu`: Τρέχουσα CPU στην οποία εκτελείται η διεργασία.
8. `mm`: Δομή που δείχνει στον χώρο διευθύνσεων της διεργασίας (virtual memory descriptor).
9. `active_mm`: Ενεργός χώρος μνήμης για την τρέχουσα διεργασία.
10. `stack`: Διεύθυνση της στοίβας του kernel για τη διεργασία.
11. `semaphore`: Πληροφορίες για τους σημαφόρους της διεργασίας.
12. `signal`: Για signal handling
13. `parent`: Δείκτης στον γονέα της διεργασίας.
14. `comm`: **όνομα της διεργασίας**
15. `files`: Δομή που αποθηκεύει τα ανοιχτά αρχεία της διεργασίας.
16. `fs`: Πληροφορίες για το file system context (current working directory, root directory).
17. `utime` και `stime`: Χρόνος χρήσης της CPU σε λειτουργία χρήστη (user mode) και kernel mode.
18. `start_time`: Ώρα εκκίνησης της διεργασίας.
19. `cred`: Πληροφορίες για τα δικαιώματα χρήστη και ομάδας (UID, GID).
20. `Δείκτες` σε άλλες διεργασίες (π.χ., child, sibling) για την υποστήριξη των δομών όπως το δέντρο διεργασιών.

Άσκηση 3:

Γράψτε ένα module πυρήνα το οποίο Θα τυπώνει πληροφορίες για όλες τις υπάρχουσες διεργασίες όταν φορτώνεται. Για να πάρουμε έναν δείκτη προς την διεργασία που εκτελείται αυτή τη στιγμή μπορούμε να χρησιμοποιήσουμε το macrocurrent.

Πηγαίνουμε στο αρχείο linux/include/linux/sched/signal.h και βρίσκουμε πώς ορίζεται το macro for_each_process:

```
#define for_each_process(p) \
        for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

Δηλαδή, διατρέχει την κυκλική διασυνδεδεμένη λίστα των διεργασιών.

Για να τυπώσουμε το όνομα της διεργασίας θα χρειαστούμε το πεδίο comm του task_struct.

Κώδικας του module **list-processes.c**:

```
#include <linux/init.h>

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched/signal.h>
MODULE_DESCRIPTION("List current processes");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");
static int my_proc_init(void)
{
struct task_struct *p; /* Needed for later **/
printk("*****************************************\n");
printk("Current process: pid = %d; name = %s\n",
current->pid, current->comm);
printk("\nProcess list:\n\n");
/* TODO */
for_each_process(p){
printk("process information:  pid: %d   name: %s",p->pid,p->comm);

}
return 0;
}
static void my_proc_exit(void)
{
printk("exiting\n");
printk("Current process: pid = %d; name = %s\n",
current->pid, current->comm);
}
module_init(my_proc_init);
module_exit(my_proc_exit);
```

Όπως και προηγουμένως, δημιουργούμε ένα Makefile:

```
obj-m + += list-processes.o
PWD := $(CURDIR)

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

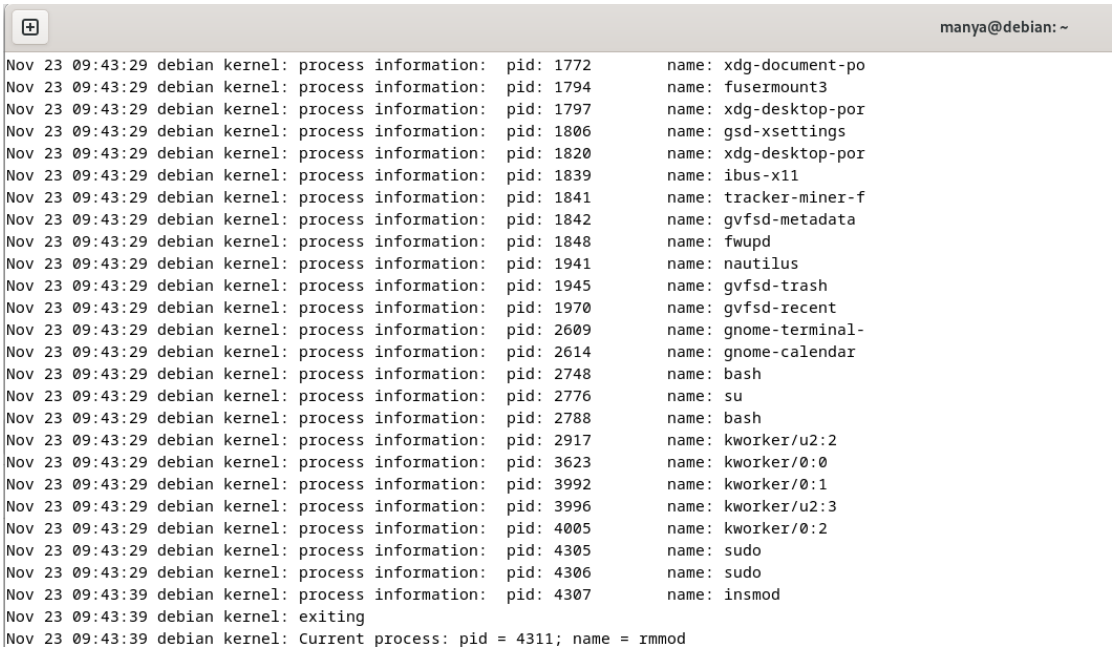
Και εκτελούμε τις εξής εντολές στο terminal:

```
root@debian:~/list-processes# make
make -C /lib/modules/6.1.0-26-amd64/build M=/root/list-processes  modules
make[1]: Entering directory '/usr/src/linux-headers-6.1.0-26-amd64'
  CC [M]  /root/list-processes/list-processes.o
  MODPOST /root/list-processes/Module.symvers
  CC [M]  /root/list-processes/list-processes.mod.o
  LD [M]  /root/list-processes/list-processes.ko
  BTF [M] /root/list-processes/list-processes.ko
Skipping BTF generation for /root/list-processes/list-processes.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-6.1.0-26-amd64'
root@debian:~/list-processes# sudo insmod list-processes.ko
root@debian:~/list-processes# sudo rmmod list-processes.ko
```

Ύστερα ελέγχουμε τι έγραψε το module στα logs:root@debian:~/list-processes# journalctl --since "10 minutes ago" | grep kernel

[…]

```
Nov 23 09:33:52 debian kernel: ****************************************
Nov 23 09:33:52 debian kernel: Current process: pid = 3976; name = insmod
Nov 23 09:33:52 debian kernel:
Nov 23 09:33:52 debian kernel: process information:  pid: 1        name: systemd
Nov 23 09:33:52 debian kernel: process information:  pid: 2        name: kthreadd
Nov 23 09:33:52 debian kernel: process information:  pid: 3        name: rcu_gp
Nov 23 09:33:52 debian kernel: process information:  pid: 4        name: rcu_par_gp
Nov 23 09:33:52 debian kernel: process information:  pid: 5        name: slub_flushwq
Nov 23 09:33:52 debian kernel: process information:  pid: 6        name: netns
Nov 23 09:33:52 debian kernel: process information:  pid: 8        name: kworker/0:0H
Nov 23 09:33:52 debian kernel: process information:  pid: 10        name: mm_percpu_wq
Nov 23 09:33:52 debian kernel: process information:  pid: 11        name: rcu_tasks_kthre
Nov 23 09:33:52 debian kernel: process information:  pid: 12        name: rcu_tasks_rude_
Nov 23 09:33:52 debian kernel: process information:  pid: 13        name: rcu_tasks_trace
Nov 23 09:33:52 debian kernel: process information:  pid: 14        name: ksoftirqd/0
Nov 23 09:33:52 debian kernel: process information:  pid: 15        name: rcu_preempt
Nov 23 09:33:52 debian kernel: process information:  pid: 16        name: migration/0
Nov 23 09:33:52 debian kernel: process information:  pid: 18        name: cpuhp/0
Nov 23 09:33:52 debian kernel: process information:  pid: 20        name: kdevtmpfs
Nov 23 09:33:52 debian kernel: process information:  pid: 21        name: inet_frag_wq
Nov 23 09:33:52 debian kernel: process information:  pid: 22        name: kauditd
Nov 23 09:33:52 debian kernel: process information:  pid: 23        name: khungtaskd
Nov 23 09:33:52 debian kernel: process information:  pid: 24        name: oom_reaper
Nov 23 09:33:52 debian kernel: process information:  pid: 27        name: writeback
Nov 23 09:33:52 debian kernel: process information:  pid: 28        name: kcompactd0
Nov 23 09:33:52 debian kernel: process information:  pid: 29        name: ksmd
Nov 23 09:33:52 debian kernel: process information:  pid: 30        name: khugepaged
Nov 23 09:33:52 debian kernel: process information:  pid: 31        name: kintegrityd
Nov 23 09:33:52 debian kernel: process information:  pid: 32        name: kblockd
Nov 23 09:33:52 debian kernel: process information:  pid: 33        name: blkcg_punt_bio
```



Λειτουργεί με τον επιθυμητό τρόπο.

Άσκηση 4:

Στην άσκηση αυτή σας ζητείται να φτιάξετε ένα kernel module το οποίο θα τυπώνει πληροφορίες για μία διεργασία και τις θυγατρικές τις διεργασίες.

Βήμα 1:

```
1038          * Children/sibling form the list of natural children:
1039          */
1040         struct list_head                children;
1041         struct list_head                sibling;
1042         struct task_struct              *group_leader;
```

- Children: Αντιπροσωπεύει μια διπλά διασυνδεδεμένη λίστα με όλα τα παιδιά (θυγατρικές διεργασίες) μιας διεργασίας, η οποία αποτελείται από κόμβους τύπου task_struct *, δηλαδή δείκτες στις δομές task_struct που αντιπροσωπεύουν κάθε παιδί.
- Sibling: Αντιπροσωπεύει τη θέση μιας διεργασίας ανάμεσα στα αδέρφια της (siblings) στη λίστα των παιδιών του γονέα της. Χρησιμοποιείται για την πλοήγηση στη λίστα των παιδιών μιας διεργασίας μέσω του γονέα.

Βήμα 2: Χρήση του macro list_for_each_entry

Το macro list_for_each_entry χρησιμοποιείται για την προσπέλαση διπλά διασυνδεδεμένων λιστών στο Linux Kernel.

```
#define list_for_each_entry(pos,head,member)        \
for (pos = list_first_entry(head, typeof(*pos), member);   \
            !list_entry_is_head(pos,head,member);          \
           pos = list_next_entry(pos,member))
```

- ptr: Μεταβλητή που δείχνει στον τρέχοντα κόμβο (συνήθως task_struct *).
- head: Η κεφαλή της λίστας (συνήθως &parent->children).
- member: Το πεδίο της δομής που συνδέει τον τρέχοντα κόμβο στη λίστα (π.χ., sibling).

Βήμα 3: Προσθέστε στο σκελετό της άσκησης τον κατάλληλο κώδικα ώστε να τυπώνεται το PID κάθε διεργασίας παιδί της αρχικής σας διεργασίας, τί παρατηρείτε;

Αρχικά, έγιναν κάποιες αλλαγές στο **forking.c**:

- Στην αρχή δίνονται 10 δευτερόλεπτα sleep, έτσι ώστε να μην αρχίσει κατευθείαν η αύξηση των διεργασιών μέσω fork και να έχουμε χρόνο να περάσουμε το pid της διεργασίας στο module (κάθε φορά μπορεί να είναι διαφορετικό).
- Στην συνέχεια υπάρχει ένας βρόχος, που στην $35^{η}$ επανάληψη θα σταματήσει (έως τότε θα έχουν δημιουργηθεί αρκετές διεργασίες-35 παιδιά της αρχικής.)
- Μιας και μας ενδιαφέρουν μόνο οι διεργασίες παιδιά της αρχικής και όχι τα παιδιά των παιδιών κ.ο.κ., αν το fork επιστρέψει 0, δηλαδή είμαστε σε child process, γίνεται break (γραμμή 22), ώστε να μην γίνει fork του παιδιού και δημιουργηθεί διεργασία παιδί του παιδιού.
- Αν το fork δεν επιστρέψει μηδέν, είμαστε δηλαδή σε διεργασία γονέα, ο βρόχος συνεχίζεται κανονικά και ξαναγίνεται fork.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#define SLEEP_TIME_LONG 10
#define SLEEP_TIME 2
int main(int argc, char *argv[])
{
    int n = 0;
    printf("PID: %ld\n", (long)getpid());
    /* Sleep for SLEEP_TIME seconds. */
    sleep(SLEEP_TIME_LONG);
    // sleep(SLEEP_TIME);
    while (1)
    {
        sleep(SLEEP_TIME);
        /* break after a few iterations (too many processes) */
        if (n++ > 35)
            exit(0);
        int f= fork();
        if (f==0){
            break;/* Child process should not fork further, so it exits the loop. */
        }
        printf("Forked! PID: %d\n", f);
    }
    return 0;
}
```

Το terminal εμφανίζει τα εξής αποτελέσματα:

```
manya@debian:/media/sf_OS3$ ./a.out
PID: 6051
Forked! PID: 6052
Forked! PID: 6055
Forked! PID: 6056
Forked! PID: 6057
Forked! PID: 6058
Forked! PID: 6059
Forked! PID: 6060
Forked! PID: 6061
Forked! PID: 6062
Forked! PID: 6063
Forked! PID: 6064
Forked! PID: 6065
Forked! PID: 6066
Forked! PID: 6067
Forked! PID: 6068
Forked! PID: 6069
Forked! PID: 6070
Forked! PID: 6071
Forked! PID: 6072
Forked! PID: 6073
Forked! PID: 6074
Forked! PID: 6075
Forked! PID: 6076
Forked! PID: 6077
Forked! PID: 6078
Forked! PID: 6079
Forked! PID: 6080
Forked! PID: 6081
Forked! PID: 6082
Forked! PID: 6083
Forked! PID: 6084
Forked! PID: 6085
Forked! PID: 6086
Forked! PID: 6087
Forked! PID: 6088
manya@debian:/media/sf_OS3$
```

Σε ξεχωριστό φάκελο στο root δημιουργείται το **Module list-children**:

Το module:

1. Θα βρίσκει το επιθυμητό process που κάθε φορά μπορεί να έχει άλλο pid. Το pid θα δίνεται μέσω της εντολής insmod list-children.ko target_pid=, χάρη στο macro `module_param(name, type, perm);`.
2. Θα βρίσκει μέσω του πεδίου children τα παιδιά της διεργασίας και θα τυπώνει πληροφορίες για αυτά.
3. Ανά χρονικό διάστημα μερικών δευτερολέπτων θα ξαναδιατρέχει την λίστα των παιδιών καθώς μπορεί να έχει αλλάξει.
4. Για ασφάλεια γίνεται χρήση της rcu_read_lock() και rcu_read_unlock(), μιας και η λίστα χρησιμοποιεί τον μηχανισμό RCU (Read-Copy-Update). Ο λόγος είναι ότι τα δεδομένα που σχετίζονται με τις διεργασίες του πυρήνα μπορούν να αλλάξουν κατά τη διάρκεια της πλοήγησης της λίστας. Η χρήση rcu_read_lock() προστατεύει από το να διαγραφούν ή να τροποποιηθούν οι διεργασίες από άλλες νήματα, καθώς ο RCU μηχανισμός διασφαλίζει ότι οι αναγνώστες μπορούν να βλέπουν τα δεδομένα σε ασφαλή κατάσταση ακόμα κι αν αυτά αλλάζουν τιμές από άλλες νήματα ή επεξεργαστές.

> `module_param(name, type, perm);`
> where *name* is the name of both the parameter exposed to the user and the variable holding the parameter inside your module. The *type* argument holds the parameter's data type; it is one of `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool`, or `invbool`. […] Finally, the `perm` argument specifies the permissions of the corresponding file in sysfs. […] A value of zero disables the sysfs entry altogether.
> Από https://litux.nl/mirror/kerneldevelopment/0672327201/ch16lev1sec6.html

Κώδικας:

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched/signal.h>
#include <linux/moduleparam.h>
#include <linux/list.h>
#include <linux/timer.h>
#include <linux/jiffies.h>
#include <linux/rcupdate.h> // Για RCU functions
MODULE_DESCRIPTION("List current process' children periodically");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");
static int target_pid = 0; /**< default Τιμή*/
static int interval = 1; /**< Χρονικό διάστημα (σε δευτερόλεπτα) */
static struct timer_list my_timer;
module_param(target_pid, int, 0); /**< Εισαγωγή του PID από terminal */
MODULE_PARM_DESC(target_pid, "The target process PID to find its
children");
void print_children(struct task_struct *task)
{
    struct list_head *list;
```

```
    struct task_struct *child;

    rcu_read_lock();
    list_for_each(list, &task->children) {
        child = list_entry(list, struct task_struct, sibling);
        printk("Child process: PID = %d, Name = %s\n", child->pid, child-
>comm);
    }
    rcu_read_unlock();
}

void my_timer_callback(struct timer_list *timer)
{
    struct task_struct *task;
    printk("***find process with pid: %d***\n", target_pid);
    rcu_read_lock();
    for_each_process(task) {
        if (task->pid == target_pid) {
            printk("Found process: PID = %d, Name = %s\n", task->pid, task-
>comm);
            print_children(task);
            break;
        }
    }
    rcu_read_unlock();
    mod_timer(&my_timer, jiffies + interval * HZ);
}

static int my_proc_init(void)
{
    struct task_struct *task;

    printk("*** find process with pid: %d ***\n", target_pid);
    rcu_read_lock();
    for_each_process(task) {
        if (task->pid == target_pid) {
            printk("Found process: PID = %d, Name = %s\n", task->pid, task-
>comm);
            print_children(task);
            break;
        }
    }
    rcu_read_unlock();
    timer_setup(&my_timer, my_timer_callback, 0);
    mod_timer(&my_timer, jiffies + interval * HZ);
    return 0;
}
static void my_proc_exit(void)
```

```
{
    del_timer(&my_timer);
    printk("Exiting module\n");
    printk("Current process: pid = %d; name = %s\n", current->pid, current-
>comm);
}
module_init(my_proc_init);
module_exit(my_proc_exit);
```

Δημιουργία Makefile όπως τις προηγούμενες φορές.

Εντολές στο terminal:
root@debian:~/list-children# sudo insmod list-children.ko target_pid=6051

root@debian:~/list-children# journalctl --since "10 minutes ago" | grep kernel

Τελικά Αποτελέσματα (το forking.c έχει ολοκληρωθεί):

Nov 30 14:58:19 debian kernel: ******** find process with pid: 6051********
Nov 30 14:58:19 debian kernel: Found process: PID = 6051, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6052, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6055, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6056, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6057, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6058, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6059, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6060, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6061, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6062, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6063, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6064, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6065, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6066, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6067, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6068, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6069, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6070, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6071, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6072, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6073, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6074, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6075, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6076, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6077, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6078, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6079, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6080, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6081, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6082, Name = a.out

Nov 30 14:58:19 debian kernel: Child process: PID = 6083, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6084, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6085, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6086, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6087, Name = a.out
Nov 30 14:58:19 debian kernel: Child process: PID = 6088, Name = a.out

Σημείωση: Στα logs υπάρχουν και εγγραφές του module Που δεν περιέχουν όλα τα παιδιά καθώς το forking.c δεν είχε ολοκληρωθεί ακόμα.

Παρατηρήσεις:
Ανιχνεύθηκαν όλα τα παιδιά. Επίσης παρατηρούμε ότι συνήθως το κάθε παιδί παίρνει το pid του προηγούμενου αυξημένο κατά 1 και το πρώτο παιδί το pid του γονέα αυξημένο κατά 1. Σε δοκιμές που έγιναν με μεγάλο αριθμό παιδιών πχ 65 αυτός ο κανόνας μπορεί να είχε κάποια εξαίρεση όπου ο αριθμός του pid να αυξανόταν κάποια στιγμή με ένα άλμα, πιθανώς επειδή το ΛΣ χρειαζόταν να δημιουργήσει και άλλες διεργασίες αναμεσά στα παιδιά του forking.c.