Λειτουργικά Συστήματα Δραστηριότητα 3

2024-2025

ΑΜ: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

Περιεχόμενα

[Άσκηση 1: 3](#_Toc183897747)

[Άσκηση 2: 4](#_Toc183897748)

[Άσκηση 3: 7](#_Toc183897749)

[Άσκηση 4: 10](#_Toc183897750)

# 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);

1. 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

1. 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

1. 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;

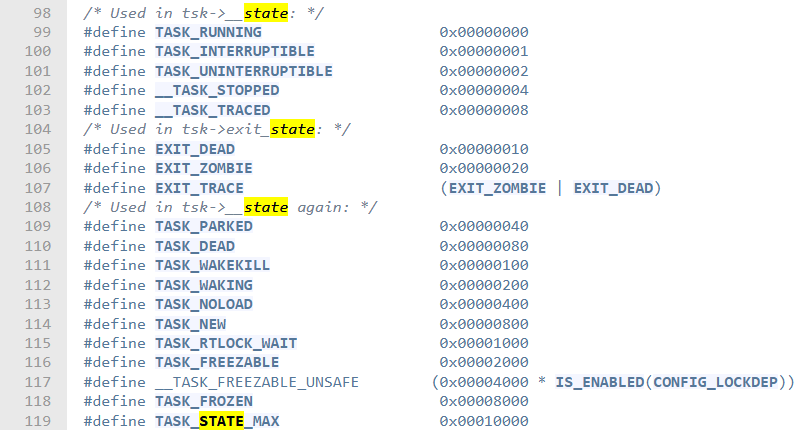
Το Linux kernel αποθηκεύει την λίστα των διεργασιών σε μια κυκλική διπλά συνδεδεμένη λίστα, που ονομάζεται **task list**. Κάθε στοιχείο αυτής της λίστας είναι τύπου **struct task\_struct.** Το struct αυτό περιγράφει μια **διεργασία, όπως το proc στο xv6** καιορίζεται στο αρχείο **<linux/sched.h>**.

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.

1. exit\_state: Κατάσταση κατά την έξοδο από τη διεργασία.

Πιθανές τιμές του exit\_state:

**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.

1. sched\_class: Πληροφορίες για την κλάση του scheduler.
2. prio και static\_prio: Προτεραιότητα της διεργασίας.
3. cpu: Τρέχουσα CPU στην οποία εκτελείται η διεργασία.
4. mm: Δομή που δείχνει στον χώρο διευθύνσεων της διεργασίας (virtual memory descriptor).
5. active\_mm: Ενεργός χώρος μνήμης για την τρέχουσα διεργασία.
6. stack: Διεύθυνση της στοίβας του kernel για τη διεργασία.
7. semaphore: Πληροφορίες για τους σημαφόρους της διεργασίας.
8. signal: Για signal handling
9. parent: Δείκτης στον γονέα της διεργασίας.
10. comm: **όνομα της διεργασίας**
11. files: Δομή που αποθηκεύει τα ανοιχτά αρχεία της διεργασίας.
12. fs: Πληροφορίες για το file system context (current working directory, root directory).
13. utime και stime: Χρόνος χρήσης της CPU σε λειτουργία χρήστη (user mode) και kernel mode.
14. start\_time: Ώρα εκκίνησης της διεργασίας.
15. cred: Πληροφορίες για τα δικαιώματα χρήστη και ομάδας (UID, GID).
16. 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

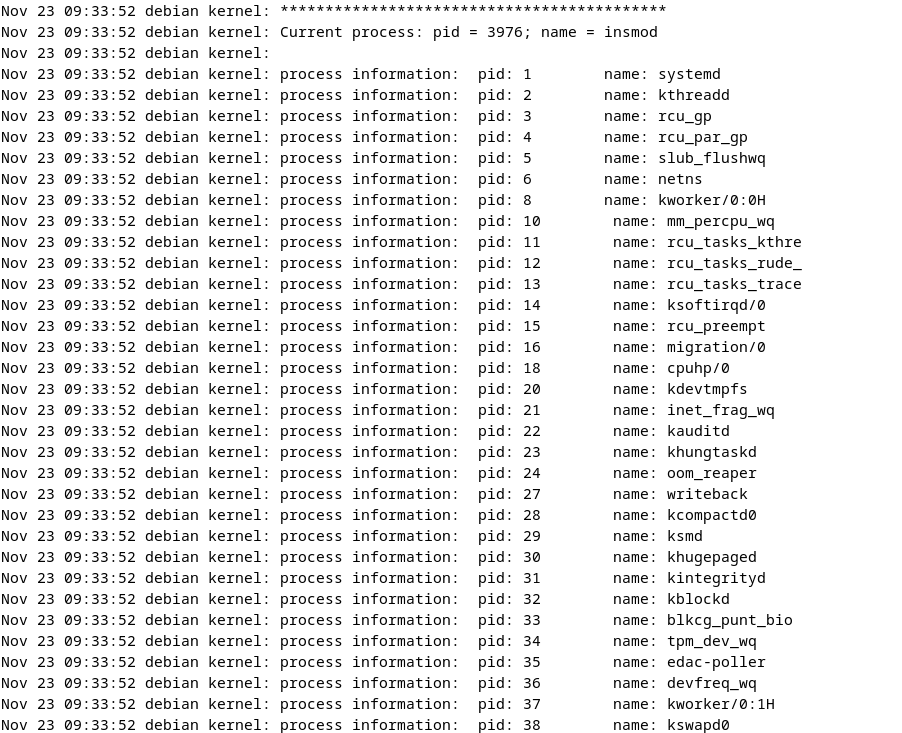
Skipping BTF generation for /root/list-processes/list-processes.ko due to unavailability of vmlinux

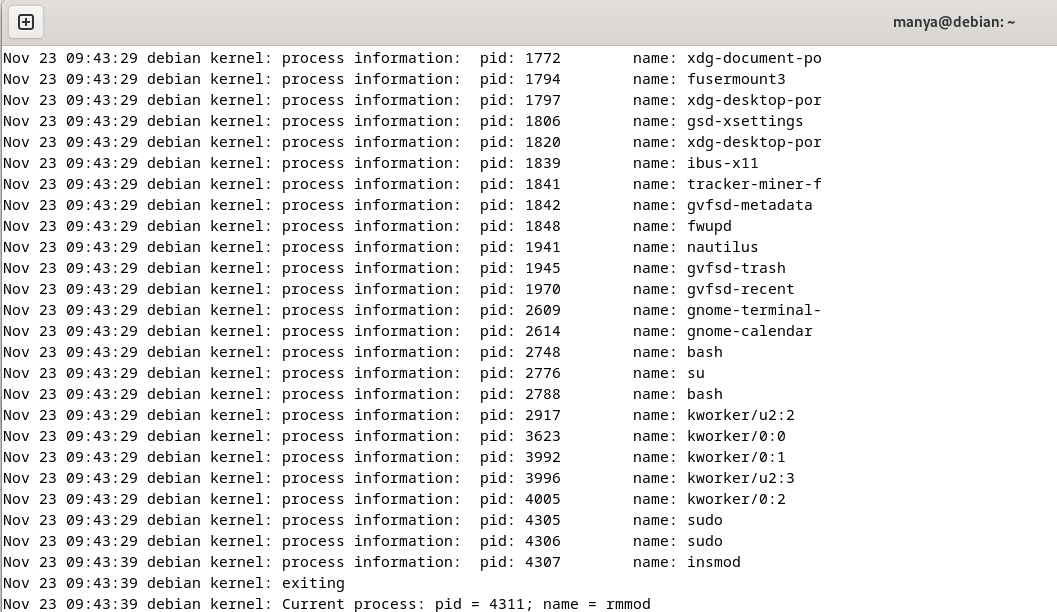
make[1]: Leaving directory '/usr/src/linux-headers-6.1.0-26-amd64'

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

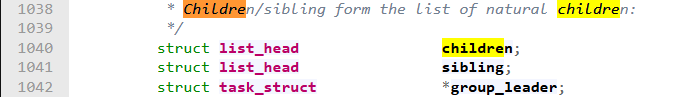
[…]



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:** 

* 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](https://elixir.bootlin.com/linux/v6.12/C/ident/list_for_each_entry)([pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos),[head](https://elixir.bootlin.com/linux/v6.12/C/ident/head),[member](https://elixir.bootlin.com/linux/v6.12/C/ident/member)) \

for ([pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos) = [list\_first\_entry](https://elixir.bootlin.com/linux/v6.12/C/ident/list_first_entry)([head](https://elixir.bootlin.com/linux/v6.12/C/ident/head), [typeof](https://elixir.bootlin.com/linux/v6.12/C/ident/typeof)(\*[pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos)), [member](https://elixir.bootlin.com/linux/v6.12/C/ident/member)); \

![list\_entry\_is\_head](https://elixir.bootlin.com/linux/v6.12/C/ident/list_entry_is_head)([pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos),[head](https://elixir.bootlin.com/linux/v6.12/C/ident/head),[member](https://elixir.bootlin.com/linux/v6.12/C/ident/member)); \

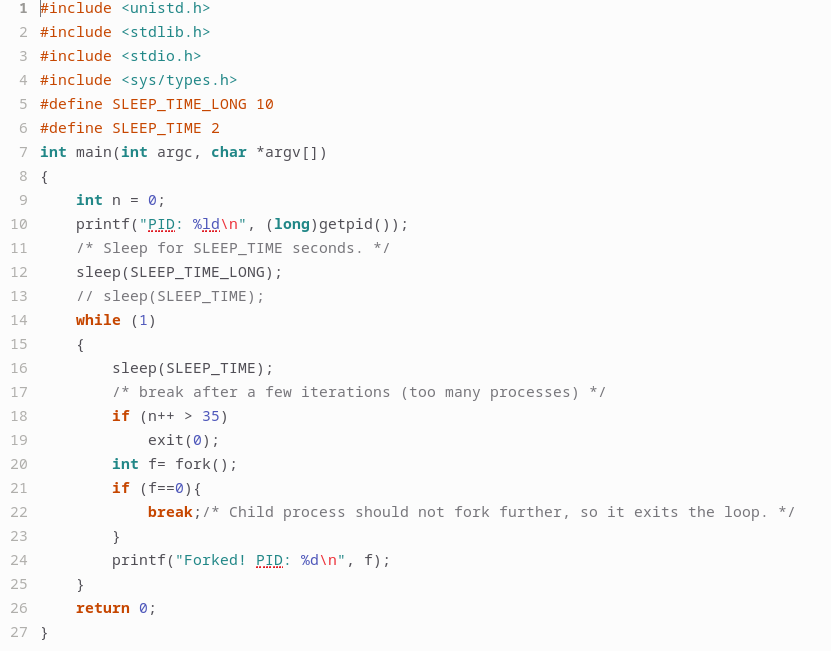
[pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos) = [list\_next\_entry](https://elixir.bootlin.com/linux/v6.12/C/ident/list_next_entry)([pos](https://elixir.bootlin.com/linux/v6.12/C/ident/pos),[member](https://elixir.bootlin.com/linux/v6.12/C/ident/member)))

* 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).

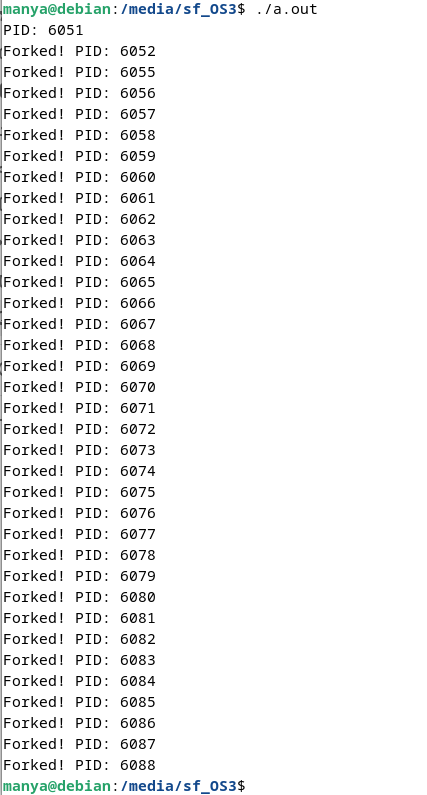
**Βήμα 3:** Προσθέστε στο σκελετό της άσκησης τον κατάλληλο κώδικα ώστε να τυπώνεται το PID κάθε διεργασίας παιδί της αρχικής σας διεργασίας, τί παρατηρείτε;

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 35η 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.



Το terminal εμφανίζει τα εξής αποτελέσματα:



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.