Λειτουργικά Συστήματα Δραστηριότητα 4

2024-2025

ΑΜ: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

Περιεχόμενα

[Ερώτημα 1: 3](#_Toc186214523)

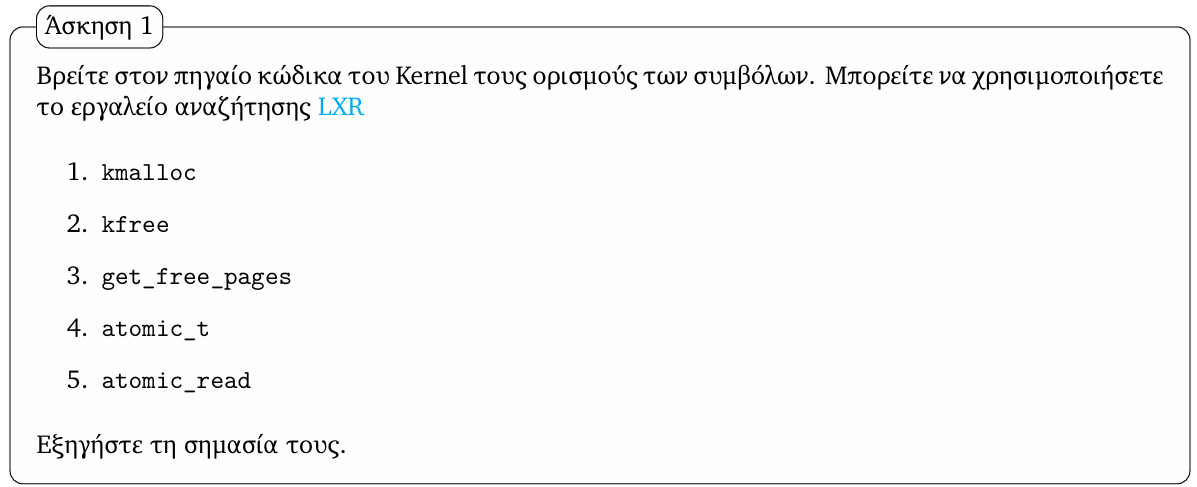
[Ερώτημα 2: 6](#_Toc186214524)

[Ερώτημα 3: 8](#_Toc186214525)

[Module έκδοση 1 χωρίς timer: 8](#_Toc186214526)

[Έκδοση 2 με timer: 11](#_Toc186214527)

# Ερώτημα 1:



Σημασία του **kmalloc**:

* definition: include/linux/slab.h
* implementation: /tools / lib / slab.c

void \***[kmalloc](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kmalloc)**(**[size\_t](https://elixir.bootlin.com/linux/v6.12.6/C/ident/size_t)** size, [**gfp\_t**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp_t) [**gfp**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp))

{

void \*ret;

if (!(**[gfp](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp)** & [**\_\_GFP\_DIRECT\_RECLAIM**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/__GFP_DIRECT_RECLAIM)))

return NULL;

ret = [**malloc**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/malloc)(size);

uatomic\_inc(&**[kmalloc\_nr\_allocated](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kmalloc_nr_allocated)**);

if (**[kmalloc\_verbose](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kmalloc_verbose)**)

[**printf**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/printf)("Allocating %p from malloc\n", ret);

if (**[gfp](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp)** & [**\_\_GFP\_ZERO**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/__GFP_ZERO))

[**memset**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/memset)(ret, 0, size);

return ret;

}

* Λειτουργία: H kmalloc επιστρέφει έναν δείκτη σε συνεχή φυσική μνήμη (resident) με το μέγεθος που ζητήσαμε. Αναλαμβάνει δηλαδή την δυναμική δέσμευση μνήμης για τον πυρήνα kernel.
* Ορίσματα: size: Μέγεθος μνήμης σε bytes, flags: Σημαίες όπως GFP\_KERNEL (για κανονική δέσμευση- regular kernel allocations) ή GFP\_ATOMIC (για περίπτωση με interrupt handlers).

Σημασία του **kfree**:

* definition: include/linux/slab.h
* implementation: /tools / lib / slab.c

void [**kfree**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kfree)(void \*p)

{

if (!p)

return;

uatomic\_dec(&**[kmalloc\_nr\_allocated](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kmalloc_nr_allocated)**);

if (**[kmalloc\_verbose](https://elixir.bootlin.com/linux/v6.12.6/C/ident/kmalloc_verbose)**)

[**printf**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/printf)("Freeing %p to malloc\n", p);

[**free**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/free)(p);

}

* Λειτουργία: H kfree αποδεσμεύει την περιοχή μνήμης του δείκτη που δέχεται σαν όρισμα.
* Ορίσματα: void\* p ο δείκτης στην συνεχή φυσική μνήμη.

Σημασία του **get\_free\_pages**:

* definition: include/linux/gfp.h

#define[**\_\_get\_free\_pages**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/__get_free_pages)(...)[**alloc\_hooks**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/alloc_hooks)([**get\_free\_pages\_noprof**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/get_free_pages_noprof)(\_\_VA\_ARGS\_\_))

* implementation: mm/page\_alloc.c

unsigned long [**get\_free\_pages\_noprof**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/get_free_pages_noprof)(**[gfp\_t](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp_t)** [**gfp\_mask**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp_mask), unsigned int [**order**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/order))

{

struct [**page**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page) \*[**page**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page);

[**page**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page) = [**alloc\_pages\_noprof**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/alloc_pages_noprof)(**[gfp\_mask](https://elixir.bootlin.com/linux/v6.12.6/C/ident/gfp_mask)** & ~[**\_\_GFP\_HIGHMEM**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/__GFP_HIGHMEM), [**order**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/order));

if (![**page**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page))

return 0;

return (unsigned long) [**page\_address**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page_address)([**page**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/page));

}

EXPORT\_SYMBOL(get\_free\_pages\_noprof);

* Λειτουργία: Ο ρόλος της είναι η δέσμευση ενός αριθμού ***σελίδων*** μνήμης για να χρησιμοποιηθούν στην kmalloc.c
* Ορίσματα: gfp\_t gfp\_mask: Το gfp\_t είναι μια μάσκα σημαίας που καθορίζει τον τρόπο και τις συνθήκες δέσμευσης μνήμης, unsigned int order: το πλήθος των σελίδων που θα δεσμευτούν

***Σημείωση:*** Οι "***σελίδες***"-pages είναι οι βασικές μονάδες μνήμης που χρησιμοποιούνται από το σύστημα διαχείρισης μνήμης στο Linux. Το μέγεθός τους είναι συνήθως 4 KB, αλλά μπορεί να ποικίλει. Σελίδα είναι η λογική ομαδοποίηση π.χ. κώδικα σε bytes, ενώ frame είναι η φυσική μνήμη. Κάθε frame-πλαίσιο περιέχει κελιά (κάθε κελί 8 bits).

Σημασία του **atomic\_t**:

* Definition και implementation: /include/linux/types.h

typedef struct {

int [**counter**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/counter);

} [**atomic\_t**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/atomic_t);

* Λειτουργία: Το atomic\_t είναι ένας τύπος δεδομένων που χρησιμοποιείται για ασφαλείς από νήματα (thread-safe) λειτουργίες σε ακέραιους αριθμούς. Διασφαλίζει ότι οι λειτουργίες (π.χ., αύξηση ή μείωση) γίνονται ατομικά (δεν υπάρχει περίπτωση διακοπής!), χωρίς τη χρήση μηχανισμών συγχρονισμού όπως locks.

Σημασία του **atomic\_read**:

* definition: include/linux/atomic/atomic-instrumented.h
* Λειτουργία: Η atomic\_read διαβάζει την τιμή μιας atomic\_t μεταβλητής με "relaxed ordering" (για γρήγορες αναγνώσεις όπου η ακριβής σειρά των λειτουργιών μνήμης δεν είναι κρίσιμη). Η instrument\_atomic\_read(v, sizeof(\*v)) χρησιμοποιείται για debugging, ενώ η πραγματική ανάγνωση γίνεται από τη raw\_atomic\_read(v).

static [**\_\_always\_inline**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/__always_inline) int

[**atomic\_read**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/atomic_read)(const [**atomic\_t**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/atomic_t) \*v)

{

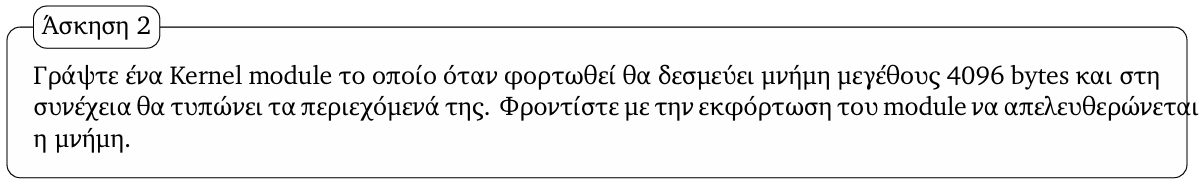
[**instrument\_atomic\_read**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/instrument_atomic_read)(v, sizeof(\*v));

return [**raw\_atomic\_read**](https://elixir.bootlin.com/linux/v6.12.6/C/ident/raw_atomic_read)(v);

}

* Ορίσματα: const atomic\_t \*v: δείκτης σε έναν τύπο δεδομένων atomic\_t

# Ερώτημα 2:



1. Δημιουργία φακέλoυ /root/contents κα δημιουργία αρχείου contents.c. Για διευκόλυνση, δημιουργία προγράμματος σε shared folder μεταξύ host και vm και ύστερα με την εντολή cp μεταφορά στο κατάλληλο αρχείο.

cp ex42.c /root/contents/contents.c

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/slab.h> // for kmalloc and kfree

MODULE\_LICENSE("GPL");MODULE\_AUTHOR("Me");

MODULE\_DESCRIPTION("OS Task 4 Exercise 2 : Memory Allocation");

#define SIZE 4096

char \*ptr;

// Module initialization

// \_\_init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers

// equivalent to initializing with my\_init, but with the added benefit of freeing the memory once the init function finishes

static int \_\_init ex42\_init(void) {

    // allocate memory

    ptr = kmalloc(SIZE, GFP\_KERNEL); // allocate memory

    // check if allocation failed

    if (ptr == NULL) {

        printk(KERN\_INFO "Memory allocation failed\n");

        return -1;

    }

    printk(KERN\_INFO "Memory allocated successfully\n");

    //print allocated memory contents

    printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    printk(KERN\_INFO "Memory contents:\n");

    printk("    DEC\t  HEX");

    int i,row = 0;

    for (i = 0; i < SIZE; i++) {

        if (i % 32 == 0){

            printk("row-%3d: ", row++);}

        if (i % 4 == 0) {

            pr\_cont("\t"); }

        pr\_cont("%x", ptr[i]); //pr\_cont prints without a newline

        }

    printk("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

    return 0;

}

**Note:** Με την εντολή cat ελέγχουμε την αντιγραφή

// Module exit

static void \_\_exit ex42\_exit(void) {

    // free the allocated memory

    kfree(ptr);

    printk(KERN\_INFO "Memory freed successfully\n");

    printk(KERN\_INFO "Exiting\n");

}

module\_init(ex42\_init);

module\_exit(ex42\_exit);

1. Δημιουργία Makefile:

obj-m += contents.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

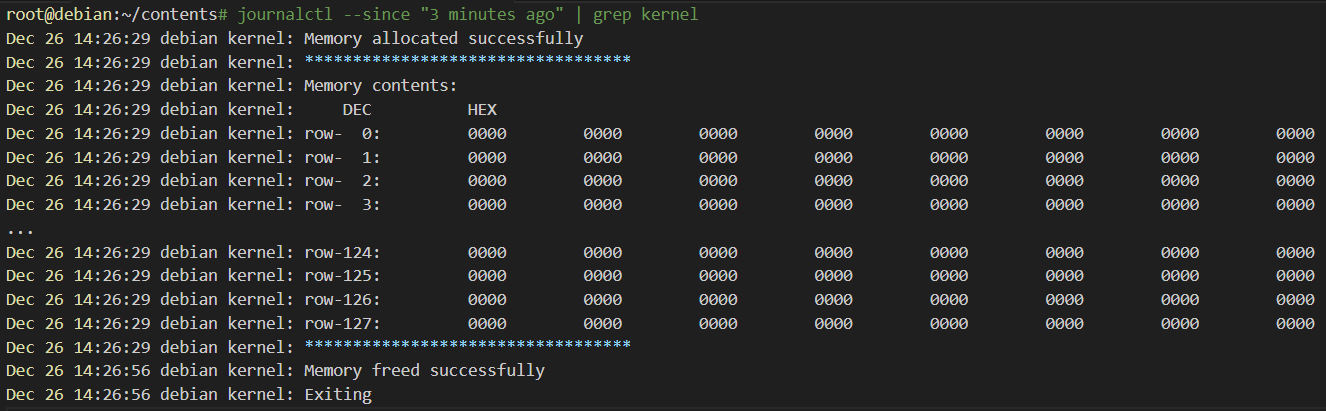
1. Εκτελούμε τις εξής εντολές στο terminal

root@debian:~/contents# make

root@debian:~/contents# sudo insmod contents.ko

root@debian:~/contents# sudo lsmod | grep contents

root@debian:~/contents# sudo rmmod contents

1. Βλέπουμε τι τυπώνει το module μας:

**Σημείωση**: Η printk τυπώνει μηνύματα στο kernel log buffer, στο οποίο μπορεί κάποιος να έχει πρόσβαση με user-space tools, όπως το journalctl.

# Ερώτημα 3:

**Κώδικας του threads.c:**

#include <unistd.h>

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define THREADS 4

void \*

thread\_func(void \*\_\_args)

{

    sleep(1);

    return NULL;

}

int main()

{

    printf("PID: %d\n", getpid());

    sleep(5);

    pthread\_t my\_threads[THREADS];

    for (int i = 0; i < THREADS; i++){

        pthread\_create(&my\_threads[i],NULL, thread\_func, NULL);}

    for (int i = 0; i < THREADS; i++){

        pthread\_join(my\_threads[i], NULL);}

}

## Module έκδοση 1 χωρίς timer:

Δημιουργούμε ένα Module που τυπώνει το mm\_users μόνο της main process.

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/sched.h>

#include <linux/sched/mm.h> // For mm\_struct access

#include <linux/moduleparam.h> // For module parameters

#include <linux/pid.h> // For PID functions

#include <linux/atomic.h> // For atomic\_t το mm\_users είναι atomic\_t

#include <linux/sched/signal.h>

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Me");

MODULE\_DESCRIPTION("OS Task 4 Exercise 3 : mm\_users μιας διεργασίας με συγκεκριμένο PID - της threads.c process");

//module parameter pid

static int pid = -1;

module\_param(pid, int, 0);

MODULE\_PARM\_DESC(pid, "Το PID της διεργασίας που θέλουμε να εξετάσουμε");

// Module initialization

static int \_\_init my\_init(void) {

    struct task\_struct \*task;

    struct mm\_struct \*mm;

    if (pid == -1) {

        printk(KERN\_ALERT "Παρακαλώ δώσε ένα έγκυρο PID\n");

        return -EINVAL;

    }

    for\_each\_process(task) {

        if (task->pid == pid) {

            mm = task->mm;

            if (!mm) {

                printk(KERN\_INFO "Η διεργασία με PID %d δεν έχει mm\_struct\n", pid);

                return 0;}

            printk(KERN\_INFO "mm\_users: %ld\n", mm->mm\_users);

            return 0;

        }

    }

    return 0;

}

// Module exit

static void \_\_exit my\_exit(void) {

    printk(KERN\_INFO "Exiting\n");

}

module\_init(my\_init);

module\_exit(my\_exit);

Κατά τα γνωστά δημιουργούμε το κατάλληλο Makefile.

Ύστερα κάνουμε compile και τρέχουμε το threads.c. Αυτό θα τυπώσει στη γραμμή εντολών το pid του. Το περνάμε ως παράμετρο στο Module μας και το φορτώνουμε στο kernel. Τέλος ελέγχουμε τα logs για να δούμε τι τύπωσε η printk:

**Terminal:**

root@debian:~/mm\_contents# sudo insmod mm\_contents.ko pid=7100

root@debian:~/mm\_contents# journalctl --since "10 minutes ago" | grep kernel

Dec 27 08:27:52 debian kernel: 09:02:24.311452 timesync vgsvcTimeSyncWorker: Radical guest time change: 33 884 812 904 000ns (GuestNow=1 735 290 144 311 444 0

Dec 27 08:27:52 debian kernel: mm\_users: 5

**Συμπέρασμα:** H mm\_users της mm\_struct φυλάει πόσα tasks (processes ή threads) μοιράζονται την ίδια μνήμη. Στην συγκεκριμένη πείπτωση η τιμή 5 είναι αναμενόμενη αφού 5 tasks (η κύρια διεργασία και τα 4 threads) μοιράζονται την ίδια μνήμη.

Κάθε thread που μοιράζεται τον ίδιο χώρο μνήμης αυξάνει τον μετρητή mm\_users κατά 1.

***Έλεγχος- Τεστ με διαφορετικό αριθμό threads:***

Για να επιβεβαιώσουμε τον παραπάνω ισχυρισμό «Κάθε thread που μοιράζεται τον ίδιο χώρο μνήμης αυξάνει τον μετρητή mm\_users κατά 1» κάνουμε τα εξής βήματα:

Αλλαγή του αριθμού threads πχ #**define** THREADS 10. Ύστερα νέο compilation και πρόσθεση του Module στο kernel με νέο Pid.

root@debian:~/mm\_contents# rmmod mm\_contents.ko

root@debian:~/mm\_contents# sudo insmod mm\_contents.ko pid=7233

root@debian:~/mm\_contents# journalctl --since "10 minutes ago" | grep kernel

Dec 27 08:42:05 debian kernel: Exiting

Dec 27 08:42:34 debian kernel: Exiting

Dec 27 08:48:36 debian kernel: mm\_users: 5

Dec 27 08:49:53 debian kernel: Exiting

Dec 27 08:50:17 debian kernel: Exiting

Dec 27 08:50:34 debian kernel: mm\_users: 11

Παρατηρούμε ότι τώρα υπάρχουν 1 (main process) + 10 (threads) = 11 χρήστες της μνήμης.

Τέλος αλλάζουμε τον αριθμό threads σε 0 #**define** THREADS 0. Επαναλαμβάνουμε τα παραπάνω βήματα (αφαιρούμε το module από τον πυρήνα και το επαναφορτώνουμε με νέο pid).

root@debian:~/mm\_contents# rmmod mm\_contents.ko

root@debian:~/mm\_contents# sudo insmod mm\_contents.ko pid=7260

root@debian:~/mm\_contents# journalctl --since "10 minutes ago" | grep kernel

Dec 27 08:48:36 debian kernel: mm\_users: 5

Dec 27 08:49:53 debian kernel: Exiting

Dec 27 08:50:17 debian kernel: Exiting

Dec 27 08:50:34 debian kernel: mm\_users: 11

Dec 27 08:54:14 debian kernel: Exiting

Dec 27 08:54:33 debian kernel: mm\_users: 1

Παρατηρούμε ότι τώρα υπάρχει μόνο 1 χρήστης, το main process όπως αναμενόμενο.

## Έκδοση 2 με timer:

Χρησιμοποιούμε το αρχείο της 8ης εβδομάδας και με μικρές παραλλαγές τυπώνουμε τα PID και mm\_users της main διεργασίας και των νημάτων. Παρατηρούμε ότι στην αρχή τυπώνει mm\_users =1 όταν υπάρχει η main διεργασία μόνο και ύστερα αφού δημιουργούνται τα threads ο αριθμός αυξάνεται σε 5 (4 thread + 1 διεργασία).

Dec 27 10:35:29 debian kernel: Found PID: 3030, name: thread!

Dec 27 10:35:29 debian kernel: ⤷ mm->mm\_users: 1

Dec 27 10:35:30 debian kernel:

Dec 27 10:35:30 debian kernel: Found PID: 3030, name: thread!

Dec 27 10:35:30 debian kernel: ⤷ mm->mm\_users: 5

Dec 27 10:35:30 debian kernel: PID: 3033, name: thread

Dec 27 10:35:30 debian kernel: PID: 3034, name: thread

Dec 27 10:35:30 debian kernel: PID: 3035, name: thread

Dec 27 10:35:30 debian kernel: PID: 3036, name: thread

Δοκιμή για 40 threads:

Dec 27 10:42:54 debian kernel: Found PID: 3404, name: thread!

Dec 27 10:42:54 debian kernel: ⤷ mm->mm\_users: 1

Dec 27 10:42:54 debian kernel:

Dec 27 10:42:54 debian kernel: Found PID: 3404, name: thread!

Dec 27 10:42:54 debian kernel: ⤷ mm->mm\_users: 41

Dec 27 10:42:54 debian kernel: PID: 3407, name: thread

Dec 27 10:42:54 debian kernel: PID: 3408, name: thread

Dec 27 10:42:54 debian kernel: PID: 3409, name: thread

Dec 27 10:42:54 debian kernel: PID: 3410, name: thread

Dec 27 10:42:54 debian kernel: PID: 3411, name: thread

Dec 27 10:42:54 debian kernel: PID: 3412, name: thread

Κώδικας:

#include "asm/processor.h"

#include "linux/list.h"

#include "linux/nodemask.h"

#include "linux/printk.h"

#include "linux/sched.h"

#include <linux/kernel.h>

#include <linux/module.h>

MODULE\_DESCRIPTION("Info about a process and it's children.");

MODULE\_AUTHOR("Me");

MODULE\_LICENSE("GPL");

static struct timer\_list check\_timer;

#define DELAY HZ/10

static int PID;

 /\* module parameters for setting the PID \*/

module\_param(PID, int, S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IWGRP);

MODULE\_PARM\_DESC(PID, "The process PID");

static void print\_process\_info(struct timer\_list \*unused)

{

    struct task\_struct \*task;

    struct task\_struct \*child;

    /\* Synchronization mechanism needed before searching for the process \*/

    rcu\_read\_lock();

    /\* Search through the global namespace for the process with the given PID \*/

    task = pid\_task(find\_pid\_ns(PID, &init\_pid\_ns), PIDTYPE\_PID);

    if (task)

    {

        printk("Found PID: %d, name: %s!", task->pid, task->comm); /\* Main process's info \*/

        printk("%\*s⤷ mm->mm\_users: %d", 8, "", atomic\_read(&task->mm->mm\_users)); /\* Memory sharing info \*/

        list\_for\_each\_entry(child, &task->thread\_group, thread\_group)

        {

            if(child->mm == task->mm)

                printk("\t  PID: %d, name: %s\n", child->pid, child->comm);

        }

        printk(" "); /\* For better formatting \*/

    }   /\* myCode \*/

    rcu\_read\_unlock(); /\* Task pointer is now invalid! \*/

    /\* Restart the timer. \*/

    check\_timer.expires = jiffies + DELAY;

    add\_timer(&check\_timer);

}

static int my\_init(void)

{

    timer\_setup(&check\_timer, print\_process\_info, 0);

    /\*

       jiffies in the variable that holds the number of ticks since the machine booted.

       We want our callback function to execute after DELAY ticks.

    \*/

    check\_timer.expires = jiffies + DELAY;

    /\* Insert the timer to the global list of active timers. \*/

    add\_timer(&check\_timer);

    return 0;

}

static void my\_exit(void)

{

    del\_timer(&check\_timer); /\* Finally, remove the timer. \*/

    printk("My Module unloaded!\n");

}

module\_init(my\_init);

module\_exit(my\_exit);

**Σύντομη Εξήγηση:**

Χρησιμοποιείται ένας kernel timer (struct timer\_list) για να εκτελεί περιοδικά τη λειτουργία print\_process\_info. Ο timer ενεργοποιείται κάθε DELAY ticks (ορίζεται ως HZ/10, δηλαδή 1 δέκατο του δευτερολέπτου). Όταν ο timer "λήξει" (expires), η print\_process\_info καλείται αυτόματα από το kernel. Η print\_process\_info αναζητά μια διεργασία με βάση το δοθέν PID(module parameter), εκτυπώνει mm\_users της διεργασίας καθώς και το όνομα της και το pid της. Ύστερα αναλύει τα threads της διεργασίας που μοιράζονται την ίδια μνήμη (τυπώνει όνομα και pid κάθε thread).