

# ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

## Δραστηριότητα 4

ΜΠΑΣΑΓΙΑΝΝΗ ΓΕΩΡΓΙΑ 1084016

ΠΑΝΟΥΡΓΙΑΣ ΑΝΤΩΝΙΟΣ 1083996

ΣΤΕΡΓΙΟΠΟΥΛΟΣ ΓΕΩΡΓΙΟΣ 1083861

### Άσκηση 1

#### kmalloc

```
/ tools / lib / slab.c
6  #include <urcu/uatomic.h>
7  #include <linux/slab.h>
8  #include <malloc.h>
9  #include <linux/gfp.h>
10
11  int kmalloc_nr_allocated;
12  int kmalloc_verbose;
13
14  void *kmalloc(size_t size, gfp_t gfp)
15  {
16      void *ret;
17
18      if (!(gfp & __GFP_DIRECT_RECLAIM))
19          return NULL;
20
21      ret = malloc(size);
22      uatomic_inc(&kmalloc_nr_allocated);
23      if (kmalloc_verbose)
24          printf("Allocating %p from malloc\n", ret);
25      if (gfp & __GFP_ZERO)
26          memset(ret, 0, size);
27      return ret;
28  }
```

Η **kmalloc** χρησιμοποιείται για δυναμική δέσμευση μνήμης στο kernel mode. Δεσμεύει σε bytes όσο χώρο καθορίζεται από το όρισμα `size` και ανάλογα με το όρισμα `gfp` εκτελεί την δέσμευση με διαφορετικό πρωτόκολλο. Είναι σημαντικό να σημειωθεί ότι οι θέσεις της μνήμης αυτής είναι συνεχόμενες σε αντίθεση με την κλασική `malloc`. Επιστρέφει δείκτη στην νέα μνήμη, αλλιώς σε περίπτωση αποτυχίας επιστρέφει `NULL`. Ορίζεται στο [/tools/lib/slab.c](#).

## kfree

```
30 void kfree(void *p)
31 {
32     if (!p)
33         return;
34     uatomic_dec(&kmalloc_nr_allocated);
35     if (kmalloc_verbose)
36         printf("Freeing %p to malloc\n", p);
37     free(p);
38 }
```

Η συνάρτηση **kfree** είναι υπεύθυνη να ελευθερώσει την προηγουμένως δεσμευμένη μνήμη από μία `kmalloc` όταν κληθεί. Είναι πολύ σημαντικό να ενημερώσουμε το λειτουργικό να απελευθερώσει τον χώρο αυτό για να μην υπάρξουν προβλήματα. Παίρνει μοναδικό όρισμα, δείκτη στο μπλοκ μνήμης που θέλουμε να ελευθερώσουμε. Ορίζεται στο </tools/lib/slab.c>.

## get\_free\_pages

```
/ tools / include / linux / types.h

18
19 typedef enum {
20     GFP_KERNEL,
21     GFP_ATOMIC,
22     __GFP_HIGHMEM,
23     __GFP_HIGH
24 } gfp_t;
```

Το **get\_free\_pages** είναι ένα enumeration που χαρακτηρίζει τα παραπάνω flags. Κάθε flag χρησιμοποιείται στην συνάρτηση `kmalloc` ώστε ανάλογα με την τιμή του να προσδιοριστεί το πρωτόκολλο υλοποίησης της δέσμευσης μνήμης, όπως να ορίσει το επίπεδο προτεραιότητας εκτέλεσής της.

## atomic\_t

```
21 typedef struct {
22     volatile int val;
23 } atomic_t;
24
```

Παρατηρούμε διάσπαρτα στον πηγαίο κώδικα διάφορους ορισμούς την **atomic\_t**, στην ουσία χρησιμοποιείται για την αναπαράσταση μιας μεταβλητής ακεραίου με ατομικές λειτουργίες. Οι ατομικές λειτουργίες ολοκληρώνουν την εκτέλεση τους χωρίς διακοπή. Συγχρονίζει την πρόσβαση σε μία μεταβλητή.

## atomic\_read

```

21  /**
22   * atomic_read - read atomic variable
23   * @v: pointer of type atomic_t
24   *
25   * Atomically reads the value of @v.
26   */
27  static inline int atomic_read(const atomic_t *v)
28  {
29      return READ_ONCE((v)->counter);
30  }
31

```

Η **atomic\_read** εξασφαλίζει ατομική πρόσβαση σε κάποιον πόρο για ανάγνωση. Λαμβάνει ως όρισμα ένα `atomic_t` που υποδεικνύει τον πόρο στον οποίο θέλουμε ατομική πρόσβαση.

## Άσκηση 2

Σε αυτή την άσκηση θέλουμε να γράψουμε ένα module με το οποίο να δεσμεύσουμε 4096 bytes μνήμης με χρήση της εντολής `kmalloс()` κατά την φόρτωσή του (`init`) και θα τυπώνει τα περιεχόμενα της μνήμης αυτής. Σημαντικό κομμάτι της υλοποίησης είναι ότι κατά την εκφόρτωση του module απελευθερώνεται η δεσμευμένη μνήμη με χρήση της `kfree()`. Βλέπουμε τον κώδικα και το αποτέλεσμα εκτέλεσής του παρακάτω:

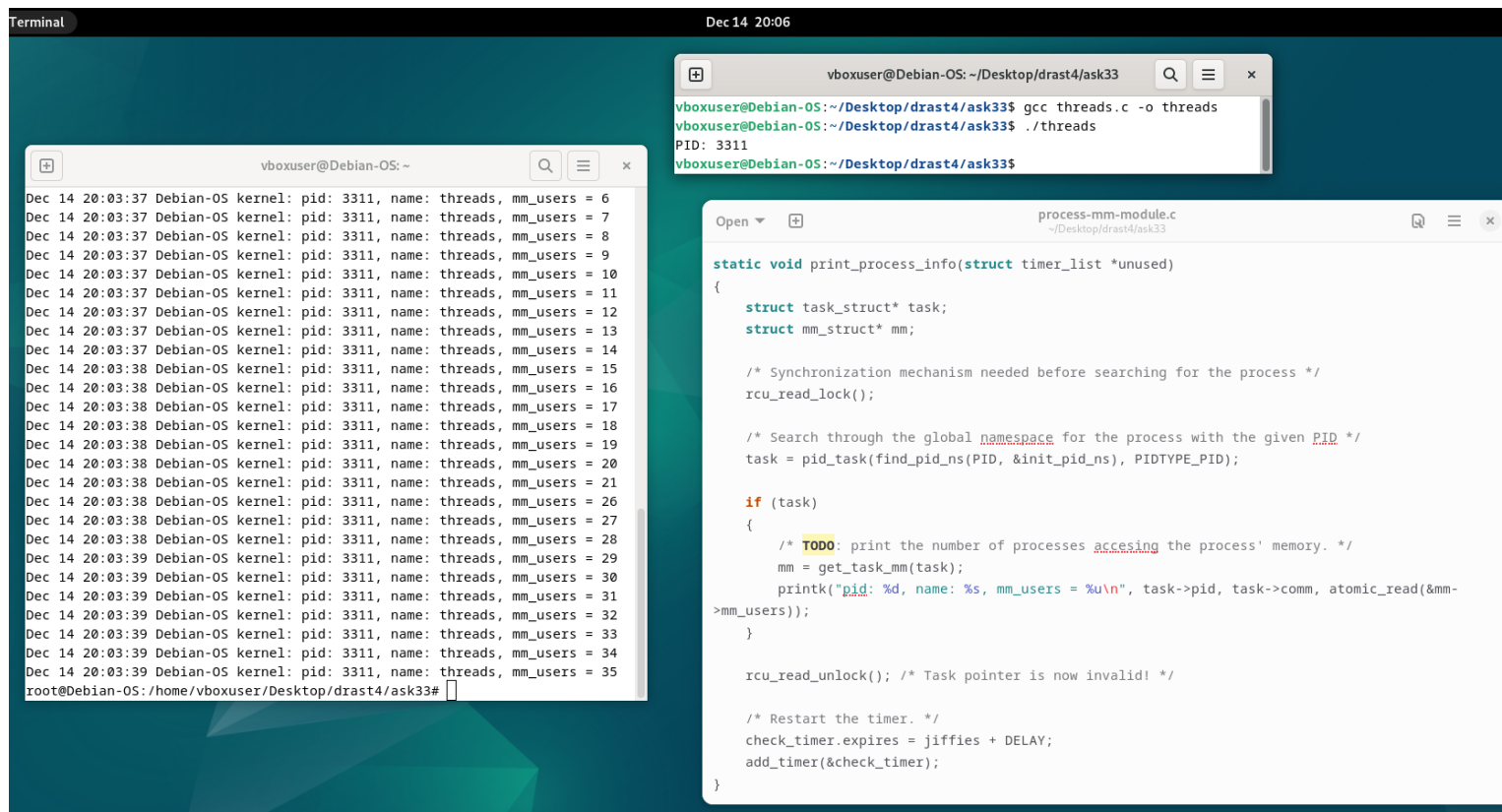
The screenshot displays a Kali Linux desktop environment. In the foreground, a terminal window is open, showing a series of log messages from the kernel. The messages indicate that memory was freed successfully, with the text "Memory freed succesfully!" appearing at the bottom. The terminal prompt is "root@Debian-OS: ~/Desktop/drast4#".

In the background, a code editor window is open, displaying the source code for a module named "ask2.c". The code includes headers for `<linux/kernel.h>`, `<linux/module.h>`, and `<linux/slab.h>`. It defines a module description, author, and license. A character array `my_memory` is declared, and a function `kali_arxi` is defined, which allocates memory using `kmalloc` and prints the memory address. A static function `kalo_telos` is also defined, which frees the memory using `kfree` and prints a message. The module initialization and exit functions are also shown.

Για έλεγχο της ορθής λειτουργίας του module μας τοποθετούμε στην τελευταία θέση της μνήμης που δεσμεύσαμε τον χαρακτήρα 'α'. Όπως φαίνεται λειτουργεί σωστά η print. Επίσης παρατηρούμε ότι όλες οι υπόλοιπες θέσεις ήταν κενές κατά την δέσμευσή τους.

### Άσκηση 3

Σε αυτή την άσκηση με χρήση κώδικα που μας δόθηκε στόχος ήταν να λάβουμε τα δεδομένα του task struct ενός process και συγκεκριμένα να τυπώσουμε την τιμή του mm\_users που μας δίνει την πληροφορία του πόσες διεργασίες μοιράζονται αυτή την θέση μνήμης. Το process το διαλέγουμε με βάση το PID που μας δίνει το πρόγραμμα threads.c και το εισάγουμε, καθώς εκτελείται το threads, στην φόρτωση του module: process-mm-module.c. Για να πάρουμε το mm του task χρησιμοποιήσαμε την συνάρτηση get\_task\_mm. Επειδή το mm->mm\_users είναι ευάλωτο σε προβλήματα διπλοεγγραφών χρησιμοποιούμε την atomic\_read() για να εξασφαλίσουμε ότι δεν έχει πρόσβαση κανείς άλλος σε αυτό κατά την διάρκεια της ανάγνωσής του. Αξίζει να σημειωθεί ότι το mm\_users είναι τύπου atomic\_t. Παρακάτω παραθέτουμε τον κώδικα στο σημείο που πραγματοποιήσαμε αλλαγές καθώς και το αποτέλεσμα εκτέλεσης του module:



The screenshot displays a development environment with a terminal window on the left and a code editor on the right. The terminal window, titled 'vboxuser@Debian-OS: ~', shows a series of kernel messages for a process with PID 3311 named 'threads'. Each message reports the value of 'mm\_users' as it increases from 6 to 35. The code editor, titled 'process-mm-module.c', shows the implementation of the 'print\_process\_info' function. This function uses 'rcu\_read\_lock()' to safely access the 'task' struct, then calls 'pid\_task()' to find the task by PID. It then uses 'get\_task\_mm()' to obtain the memory descriptor 'mm' and 'atomic\_read()' to safely read the 'mm\_users' value. A 'TODO' comment indicates the purpose of the print statement. Finally, it calls 'rcu\_read\_unlock()' and updates a timer.

```
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 6
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 7
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 8
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 9
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 10
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 11
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 12
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 13
Dec 14 20:03:37 Debian-OS kernel: pid: 3311, name: threads, mm_users = 14
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 15
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 16
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 17
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 18
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 19
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 20
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 21
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 26
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 27
Dec 14 20:03:38 Debian-OS kernel: pid: 3311, name: threads, mm_users = 28
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 29
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 30
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 31
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 32
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 33
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 34
Dec 14 20:03:39 Debian-OS kernel: pid: 3311, name: threads, mm_users = 35
root@Debian-OS: /home/vboxuser/Desktop/drast4/ask33#
```

```
static void print_process_info(struct timer_list *unused)
{
    struct task_struct* task;
    struct mm_struct* mm;

    /* 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)
    {
        /* TODO: print the number of processes accessing the process' memory. */
        mm = get_task_mm(task);
        printk("pid: %d, name: %s, mm_users = %u\n", task->pid, task->comm, atomic_read(&mm-
>mm_users));
    }

    rcu_read_unlock(); /* Task pointer is now invalid! */

    /* Restart the timer. */
    check_timer.expires = jiffies + DELAY;
    add_timer(&check_timer);
}
```

Παρατηρούμε ότι η τιμή του mm\_users αυξάνεται καθώς το module μας καλεί την συνάρτηση αυτή πολλές φορές δημιουργώντας νέα processes σε αυτόν τον χώρο.