Λειτουργικά Συστήματα Δραστηριότητα 4

2024-2025

AM: 1096060

Ονοματεπώνυμο: Μαρία-Νίκη Ζωγράφου

Περιεχόμενα

Ερώτημα 1:

Άσκηση 1

Βρείτε στον πηγαίο κώδικα του Kernel τους ορισμούς των συμβόλων. Μπορείτε να χρησιμοποιήσετε το εργαλείο αναζήτησης LXR

1. `kmalloc`
2. `kfree`
3. `get_free_pages`
4. `atomic_t`
5. `atomic_read`

Εξηγήστε τη σημασία τους.

Σημασία του **kmalloc**:

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

```
void *kmalloc(size_t size, gfp_t gfp)
{
        void *ret;

        if (!(gfp & __GFP_DIRECT_RECLAIM))
                return NULL;

        ret = malloc(size);
        uatomic_inc(&kmalloc_nr_allocated);
        if (kmalloc_verbose)
                printf("Allocating %p from malloc\n", ret);
        if (gfp & __GFP_ZERO)
                memset(ret, 0, size);
        return ret;
}
```

- Λειτουργία: Η 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(void *p)
{
        if (!p)
                return;
        uatomic_dec(&kmalloc_nr_allocated);
        if (kmalloc_verbose)
                printf("Freeing %p to malloc\n", p);
        free(p);
}
```

- Λειτουργία: Η 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(...)alloc_hooks(get_free_pages_noprof(__VA_ARGS__))
```

- implementation: mm/page_alloc.c

```
unsigned long get_free_pages_noprof(gfp_t gfp_mask, unsigned int order)
{
        struct page *page;

        page = alloc_pages_noprof(gfp_mask & ~__GFP_HIGHMEM, order);
        if (!page)
                return 0;
        return (unsigned long) page_address(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;
} 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

```
static __always_inline int
atomic_read(const atomic_t *v)
{
        instrument_atomic_read(v, sizeof(*v));
        return raw_atomic_read(v);
}
```

- 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)`.
- Ορίσματα: `const atomic_t *v`: δείκτης σε έναν τύπο δεδομένων atomic_t

Ερώτημα 2:

Άσκηση 2

Γράψτε ένα Kernel module το οποίο όταν φορτωθεί θα δεσμεύει μνήμη μεγέθους 4096 bytes και στη συνέχεια θα τυπώνει τα περιεχόμενά της. Φροντίστε με την εκφόρτωση του module να απελευθερώνεται η μνήμη.

1. Δημιουργία φακέλου /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`

```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;
}
```

```c
// 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);
```

Note: Με την εντολή cat ελέγχουμε την αντιγραφή

2. 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
```

3. 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

4. Βλέπουμε τι τυπώνει το module μας:

```
root@debian:~/contents# journalctl --since "3 minutes ago" | grep kernel
Dec 26 14:26:29 debian kernel: Memory allocated successfully
Dec 26 14:26:29 debian kernel: ********************************
Dec 26 14:26:29 debian kernel: Memory contents:
Dec 26 14:26:29 debian kernel:     DEC         HEX
Dec 26 14:26:29 debian kernel: row-  0:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-  1:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-  2:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-  3:        0000       0000       0000       0000       0000       0000       0000       0000
...
Dec 26 14:26:29 debian kernel: row-124:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-125:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-126:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: row-127:        0000       0000       0000       0000       0000       0000       0000       0000
Dec 26 14:26:29 debian kernel: ********************************
Dec 26 14:26:56 debian kernel: Memory freed successfully
Dec 26 14:26:56 debian kernel: Exiting
```

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:

Άσκηση 3

Φτιάξτε ένα kernel module το οποίο θα βρίσκει το `task_struct *` που θα αντιστοιχεί σε μία διεργασία με ένα δοθέντο `PID` και εξετάστε το `mm` μέλος του `task_struct *`. Τυπώστε το πεδίο του `mm_users`. Αυτό περιέχει τον αριθμό των διεργασιών που μοιράζονται αυτή τη θέση μνήμης. Βασιστείτε στον κώδικα που δίνεται στο αρχείο `process-mm-module.c`.

Ξεκινήστε μία διεργασία η οποία θα δημιουργεί μερικά threads. Μπορείτε να βασιστείτε στον παρακάτω κώδικα.

Τί παρατηρείτε όταν το module σας τυπώνει την τιμή του πεδίου `task->mm->mm_users`?

Κώδικας του threads.c:

```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.

```c
#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
```

```c
#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: Η 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:

```c
#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 */
```

```c
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).