➔ 1)

1. Kmalloc: Ο ορισμός της kmalloc βρίσκεται μέσα στο αρχείο slab.h, όπου βρίσκονται και όλες οι ειδικές συναρτήσεις τις οποίες η «γενική» kmalloc μπορεί να καλέσει, όπως η kmalloc_large, kmalloc_index κτλ, πολλές από τις οποίες γίνονται implemented στο αρχείο slab.c

```
590  static __always_inline __alloc_size(1) void *kmalloc(size_t size, gfp_t flags)
591  {
592          if (__builtin_constant_p(size) && size) {
593                  unsigned int index;
594
595                  if (size > KMALLOC_MAX_CACHE_SIZE)
596                          return kmalloc_large(size, flags);
597
598                  index = kmalloc_index(size);
599                  return kmalloc_trace(
600                                  kmalloc_caches[kmalloc_type(flags, _RET_IP_)][index],
601                                  flags, size);
602          }
603          return __kmalloc(size, flags);
604  }
```

Ο ρόλος της kmalloc είναι να επιστρέψει έναν δείκτη σε resident (συνεχή φυσική) μνήμη με το μέγεθος που ζητήσαμε.

2. kfree: Ο ορισμός της kfree βρίσκεται και αυτός στο αρχείο slab.h και ο ρόλος τής είναι παρόμοιος με της free στην Clib, αποδεσμεύει την περιοχή (εδώ resident) μνήμης που δέσμευε μέχρι πρότινος ο δείκτης που δέχεται.

```
227  void kfree(const void *objp);
228  void kfree_sensitive(const void *objp);
229  size_t __ksize(const void *objp);
230
231  DEFINE_FREE(kfree, void *, if (_T) kfree(_T))
```

3. get_free_pages: Βρίσκουμε τον ορισμό στο αρχείο page_alloc.c

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

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

Ο ρόλος της είναι η δέσμευση ενός αριθμού σελίδων μνήμης για να χρησιμοποιηθούν στην kmalloc.

4. atomic_t: Ο ορισμός (μέσω typedef) γίνεται στο αρχείο types.h

```
171
172   typedef struct {
173           int counter;
174   } atomic_t;
175
```

Ο ρόλος αυτής της δομής είναι να διασφαλίζει ότι τα όποια operations γίνονται σε αυτήν την μεταβλητή δεν διακόπτονται, οπότε δεν έχουμε προβλήματα ταυτοχρονισμού.

5. atomic_read: Ο ορισμός βρίσκεται στο atomic-instrumented.h

```
28    */
29   static __always_inline int
30   atomic_read(const atomic_t *v)
31   {
32           instrument_atomic_read(v, sizeof(*v));
33           return raw_atomic_read(v);
34   }
35
```

Βλέπουμε ότι διαβάζει την τιμή της μεταβλητής της δομής διασφαλίζοντας ότι δεν θα έχουμε πρόβλημα ταυτοχρονισμού στην ανάγνωσή της.

➔ 2)

Δημιουργούμε κατά τα γνωστά το Makefile και…

```
 GNU nano 7.2                          Makefile
obj-m += memory.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
```

… το αρχείο memory.c από το οποίο θα δημιουργήσουμε το module που θα φορτωθεί στο kernel.

Με την χρήση της εντολής kmalloc δεσμεύουμε μνήμη μεγέθους 4096 bytes.

Στην συνέχεια ελέγχουμε αν έγινε επιτυχής δέσμευση και τυπώνουμε τα 4096 byte αυτά ερμηνεύοντας τα ως ακεραίους των 4 bytes.

Η εντολή my_ptr[1]=55 έγινε απλά για παραστατικούς λόγους.

Σημαντικό βήμα είναι η αποδέσμευση της μνήμης με την εκφόρτωση του module για να μην δημιουργούμε memory leaks.

```
 GNU nano 7.2                          memory.c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>

MODULE_DESCRIPTION("My memory allocating kernel module");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");

int * my_ptr;

static int my_init(void)
{
        int i;
        my_ptr = kmalloc(4096, GFP_KERNEL);
        if (!my_ptr){printk("Failed to allocate!\n"); return 0;};
        printk("Allocated 4096 bytes!\n");
        printk("Printing the contents of the memory segment interpreted as 4byte integ>
        my_ptr[1]=55;
        for(i=0; i<(int)(4096/4); i++)
        {
```

```
 GNU nano 7.2                          memory.c
        printk("Printing the contents of the memory segment interpreted as 4byte integ
        my_ptr[1]=55;
        for(i=0; i<(int)(4096/sizeof(int)); i++)
        {
                printk("%d\n", my_ptr[i]);
        }
        printk("Done!\n");

        return 0;
}

static void my_exit(void)
{
        kfree(my_ptr);
        printk("Released 4096 bytes!\n");
}

module_init (my_init );
module_exit (my_exit );
```

Και έχουμε το εξής αποτέλεσμα (στα logs του kernel):

```
Dec 03 21:17:07 debian kernel: Allocated 4096 bytes!
Dec 03 21:17:07 debian kernel: Printing the contents of the memory segment interpreted
as 4byte integers
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 55
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
```

...

```
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: 0
Dec 03 21:17:07 debian kernel: Done!
Dec 03 21:17:12 debian kernel: Released 4096 bytes!
root@debian:~/memory#
```

(Μεγάλο πλήθος γραμμών έχει προφανώς αγνοηθεί, παρ'όλα αυτά όλες η γραμμές ήταν 0)

Το γεγονός ότι το τμήμα μνήμης που δεσμεύσαμε ήταν όλο αρχικοποιημένο σε 0 δεν μας εκπλήσσει ιδιαίτερα αφού το να είναι το κομμάτι μνήμης αρχικοποιημένο σε μηδενικά (ενώ δεν είναι διασφαλισμένο) είναι το πιο πιθανό σενάριο.

➔ 3)

Αρχικά αυξάνουμε τον χρόνο αναμονής κάθε νήματος ώστε να προλάβουμε να τρέξουμε τις απαραίτητες εντολές (στο αρχείο threads.c):

```
 GNU nano 7.2                                threads.c
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define THREADS 4

void* thread_func(void* __args) {
    sleep(30);
    return NULL;
}
```

Στην συνέχεια προσθέτουμε τον κώδικα που θα τυπώνει το πλήθος των αναφορών στην μνήμη της διεργασίας.

```
   /**
    * @mm_count: The number of references to &struct
    * mm_struct (@mm_users count as 1).
(   *                                                  )
```

```
if (task)
{

    printk("pid: %d, name: %s\n", task->pid, task->comm);
    printk("  mm_users: %ld\n", task->mm->mm_users);
}
```

(Παρόλο που η μεταβλητή είναι atomic_t καταχρηστικά την τυπώνουμε σαν long integer).

Αφού τρέξουμε το αρχείο threads και τυπώσει το PID της διεργασίας

```
root@debian:~/mm# ./threads
PID: 24766
```

Κάνουμε insert το module δίνοντας σαν παράμετρο το PID της διεργασίας που μόλις δημιουργήθηκε.

Αποτέλεσμα:

```
make[1]: Leaving directory '/usr/src/linux-headers-6.1.0-13-amd64'
root@debian:~/mm# sudo insmod process-mm-module.ko PID=24766
root@debian:~/mm# sudo rmmod process-mm-module
root@debian:~/mm# journalctl --since "2 minutes ago" | grep kernel
Dec 08 22:09:05 debian kernel: pid: 24766, name: threads
Dec 08 22:09:05 debian kernel:   mm_users: 5
```

Ερμηνεύοντας το αποτέλεσμα βλέπουμε ότι (όπως είναι λογικό) υπάρχουν 5 αναφορές στην μνήμη που έχει διατεθεί στην διεργασία, αυτές οι αναφορές ανήκουν στην ίδια την διεργασία και τα 4 νήματά της.