REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Memory management

1.1 kmalloc()

For simple allocation/freeing memory in kernel space, one would use:

```
#include <linux/slab.h>

void *kmalloc (unsigned int len, gfp_t gfp_mask);
void kfree (void *ptr);
```

where gfp_mask is normally set to one of the following:

GFP_KERNEL	Block and cause going to sleep if the memory is not immediately available, allowing preemption to occur. This is the normal way of calling kmalloc().
GFP_ATOMIC	Return immediately if no pages are available. For instance, this might be done when kmalloc() is being called from an interrupt, where sleep would prevent receipt of other interrupts.
GFP_DMA	For buffers to be used with ISA DMA devices, is OR'ed with GFP_KERNEL or GFP_ATOMIC. Ensures the memory will be contiguous and falls under MAX_DMA_ADDRESS=16MB for ISA devices; for PCI this is unnecessary. The exact meaning of this flag is platform dependent.

Like malloc() in user space, kmalloc will return the base address of the piece of kernel memory with at least the size of len, and on failure, return NULL.

The in_interrupt() macro can be used to check whether you are in an interrupt context; and similarly, in_atomic() also check if you are in a preemptible context.

For example:

```
char *buffer = kmalloc(nbytes, in_interrupt() ? GFP_ATOMIC : GFP_KERNEL);
```

In this example, if it is in the interrupt context, GFP_ATOMIC mode will be used, otherwise, the less expensive GFP_KERNEL mode will be used instead.

Since GFP_ATOMIC is allowed to take more memory resources than GFP_KERNEL to lessen chances of failure, therefore GFP_ATOMIC should not be used unless necessary.

Memory allocated by kmalloc() can be resized by:

```
void *krealloc(const void *p, size_t new_size, gfp_t flags);
```

kzalloc():

```
void *kzalloc(size_t size, gfp_t flags);
```

works like kmalloc, but also zero the memory.

kmalloc() will return a memory chunk with size of power of 2 that matches or exceeds len and will return NULL upon failure. The maximum size allocatable by kmalloc() is 1024 pages, or 4MB on x86. Generally for requests larger than 64kB, one should use __get_free_page() functions to ensure inter-platform compatibility.

1.2 __get_free_pages()

To allocate (and free) entire pages (or multiple pages) at once, one can use:

```
#include <linux/mm.h>

unsigned long get_zeroed_page (gfp_t gfp_mask);
unsigned long __get_free_page (gfp_t gfp_mask);
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned long order);

void free_page (unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

__get_free_page() returns the base address of a page in kernel space, that can be used directly by the kernel. The gfp_mask is the same as kmalloc()

__get_free_pages() is like get_free_page(), but allocates consecutive pages in kernel space. order gives the number of pages in the power of 2. The maximum number of pages allocatable by get_free_pages is 1024 (i.e. order = 10). Here is an example:

```
/* allocates 2 ^ 5 = 32 pages of kernel memory */
tty->read_buf = (unsigned char *)__get_free_pages(in interrupt() ? GFP_ATOMIC : GFP_KERNEL, ↩
     5);

if (!tty->read_buf) return -ENOMEM;
ALLOC_PAGE()
```

In cases where you might need to deal with high memory, which does not have a constant address in kernel space, you will need to deal with struct page directly. For example in assignment 1, you will need to get the page frame number out of each page allocated in order to map the page with mmap(). In these cases, you should allocate and free the page using:

```
struct page *alloc_page(unsigned int flags);
void __free_page(struct page *page);
```

alloc_page() works like __get_free_pages(), except the internal kernel structure struct page is returned instead of the kernel base address of the page.

Here is an example showing how to add a page to your page list:

```
/**
 *  The node structure for the memory page linked list.
 *
 */
typedef struct node {
    struct list_head list;
    struct page *asgn1_page;
} NODE;

struct list_head mem_list;


size_t foo() {
    size_t size_written = 0;
    struct list_head *ptr = mem_list.next;
.....
    NODE *curr;
.....
    if (ptr == &mem_list) {
        /* need to add a page to the page list of your module */
        curr = kmalloc(sizeof(NODE), GFP_KERNEL);

        /* check curr is not NULL..... */
```

```
        curr->asgn1_page = alloc_page(GFP_KERNEL);

        if (NULL == curr->asgn1_page) {
            printk(KERN_WARNING "Not enough memory left\n");
            return size_written;
        }
        list_add_tail(&(curr->list), &mem_list);
        num_pages++;
        ptr = mem_list.prev;
    }
....
    return size_written;
}
```

To get the page frame number from a struct page you can use the function:

```
unsigned long page_to_pfn(struct page *page);
```

and to get the kernel virtual address of a page, you can use:

```
void *page_address(struct page *page);
```

Here is an example showing how to copy data from user space to a page:

```
ssize_t foo(const char __user* buf, size_t count) {
    size_t size_to_copy = min(count, PAGE_SIZE);
    size_t size_not_copied;
    struct page *my_page = alloc_page(GFP_KERNEL);


    /* check for errors ... */


    /* copy from user space to the beginning of the
       page we have just allocated */
    size_not_copied =  copy_from_user(page_address(my_page), buf, size_to_copy);

    .......
}
```

1.3 vmalloc()

vmalloc() allocates a contiguous memory region in the virtual address space. This is the API for allocating and freeing memory with the vmalloc approach:

```
#include <linux/vmalloc.h>

void *vmalloc(unsigned long size);
void vfree(void *ptr);
```

vmalloc() returns a pointer to a linear memory area of size at least **size**, returns NULL if an error occurs.

vmalloc() cannot be used when real physical address is needed (such as for DMA) and cannot be used at interrupt time. vmalloc() can allocate more memory than get_free_pages() because the allocated memory may not be consecutive in physical memory, but the kernel sees see the memory as a contiguous range of addresses, The resulting virtual addresses are higher than the top of the physical memory.

vmalloc() has a higher overhead than get_free_pages(), so vmalloc() should not be used for small requests. Here is a code example:

```
in_buf[dev] = (struct mbuf *)vmalloc(sizeof(struct mbuf));

if (in_buf[dev] = NULL) {
  printk(KERN_WARNING "Can't allocate buffer in_buf\n");
  my_dev[dev]->close(dev);
  return -EIO;
}
```

Note we do not recommend you to use vmalloc() in our memory devices.

1.4 slabs and cache allocations

If you want to allocate memory for objects less than a page in size and you don't want to waste space by requesting whole pages, and you need to create and destroy such objects (with the same size) frequently, it would be more efficient to allocate your own pool of memory and set up your own caching system, rather than repeatedly allocating/freeing these objects through the linux kmalloc() system. The Linux has set up the slab allocator interface that you should use. As part of the scheme, you can create a special memory pool, add and remove objects. The kernel can dynamically shrink the cache if it has memory needs elsewhere, but it will not have to re-allocate a new object every time you need one, as long as there are still wholly- or partially-unused slabs on the cache.

The following functions can create, shrink and destroy your own memory cache:

```
#include <linux/slab.h>

struct kmem_cache *kmem_cache_create(const char *name, size_t size,
                                     size_t offset, unsigned long flags,
                                     void (*ctor)(void *,struct kmem_cache *, unsigned long ↩
                                          flags));

int kmem_cache_shrink(struct kmem_cache *cache);

void kmem_cache_destroy(struct kmem_cache *cache);
```

where **name** serves to identify the cache in the system, viewable in /proc/slabinfo. All objects in the cache must have the same size, which is specified by the parameter size that cannot be more than 1024 pages (4MB on x86).

offset indicates alignment, or offset into the page for the objects you are allocating (normally set to **0**).

The flags argument is a bitmask of choices given in /usr/src/linux/include/linux/slab.h

SLAB_HWCACHE_ALIGN	Force alignment of data objects on cache lines. Improves performance but may waste memory. Should be set for critical performance code.
SLAB_POISON	Fill the slab layer with the known value a5a5a5a5. Good for catching access to uninitialized memory.
SLAB_RED_ZONE	Surround allocated memory with red zones that scream when touched, to detect buffer overruns.
SLAB_PANIC	Causes system panic upon allocation failure.
SLAB_DEBUG_FREE	Perform expensive checks on freeing objects
SLAB_CACHE_DMA	Make sure the allocation is in the DMA zone.

ctor is an optional constructor function used to initialize any objects before they are used. If not provided, this parameter is set to NULL.

To view slab caches currently allocated in your system, you can use **slabtop** or **vmstat -m**. Read their manpage entries for more information.

Once the cache is set up, you can allocate / free objects by:

```
void *kmem_cache_alloc(struct kmem_cache *cache, gfp_t gfp_mask);
void kmem_cache_free(struct kmem_cache *cache, void *);
```

You can use the function kmem_cache_shrink() to release unused objects. When you finish using the memory cache, you must free it up by calling kmem_cache_destroy(), otherwise resources will not be freed. This function will fail if there are still objects in use.

For example:

```
#include <linux/module.h>
#include <linux/slab.h>

static int size = PAGE_SIZE;
static struct kmem_cache *my_cache;
module_param(size, int, S_IRUGO);

static int mycdrv_open(struct inode *inode, struct file *file)
{
    /* allocate a memory cache object */

    if (!(ramdisk = kmem_cache_alloc(my_cache, GFP_ATOMIC))) {
        printk(KERN_ERR " failed to create a cache object\n");
        return -ENOMEM;
    }
    printk(KERN_INFO " successfully created a cache object\n");
    return mycdrv_generic_open(inode, file);
}

static int mycdrv_release(struct inode *inode, struct file *file)
{
    /* destroy a memory cache object */
    kmem_cache_free(my_cache, ramdisk);
    printk(KERN_INFO "destroyed a memory cache object\n");
    printk(KERN_INFO " closing character device: %s:\n\n", MYDEV_NAME);
    return 0;
}

static int __init my_init(void)
{
    /* create a memory cache with blocks of specific size */

    if (size > (1024 * PAGE_SIZE)) {
        printk
            (KERN_INFO
             " size=%d is too large; you can't have more than 1024 pages!\n",
             size);
        return -1;
    }

    if (!(my_cache = kmem_cache_create("mycache", size, 0,
                        SLAB_HWCACHE_ALIGN, NULL))) {
        printk(KERN_ERR "kmem_cache_create failed\n");
        return -ENOMEM;
    }
    printk(KERN_INFO "allocated memory cache correctly\n");
    ramdisk_size = size;

.........
    return 0;
}

static void __exit my_exit(void)
```

```
{
........
    (void)kmem_cache_destroy(my_cache);
}

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

1.4.1 Exercise: memory caches

In one of the modules you previously created that uses kmalloc(), implement memory cache and make one of the kmalloc() calls use your memory cache instead. Make sure you free any slabs you create.

1.4.2 Exercise: Testing maximum memory allocation (optional)

See how much memory you can obtain dynamically, using both kmalloc() and get_free_pages().

Start with requesting 1 page of memory, and then keep doubling until your request fails for each type fails. Make sure you free any memory you receive.

You will probably want to use GFP_ATOMIC rather than GFP_KERNEL. (why?)

If you have trouble getting enough memory due to memory fragmentation trying writing a poor-man's de-fragmenter, and then running again. The de-fragmenter can just be a module that grabs all available memory, use it and then release it when done, thereby clearing the caches. You can also try the command:

```
$ sync; echo 3 > /proc/sys/vm/drop_caches
```

Try the same thing with vmalloc() Rather than doubling allocations, start at 4MB and increases in 4MB increments until failure results. Note this may hang while loading. (Why?)

Kernel code cannot directly access user-space memory and user-space programs also cannot directly access kernel memory. To transfer data between user and kernel spaces, one must use copying constructs, or memory mapping.

1.5 copying data across user/kernel space

To copy the value of a variable from the user space to kernel space, one can use:

```
#include <linux/uaccess.h>

int get_user(lvalue, ptr);
```

which copy value pointed by **ptr** at user space to the kernel space variable **lvalue**. Returns **0** for success, **-EFAULT** otherwise.

Here is an example which data from the user space is copied to the kernel memory byte by byte:

```
static inline ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
             loff_t * ppos)
{
    int nbytes = 0, maxbytes, bytes_to_do;
    char *tmp = ramdisk + *ppos;
    maxbytes = ramdisk_size - *ppos;
    bytes_to_do = maxbytes > lbuf ? lbuf : maxbytes;
    if (bytes_to_do == 0)
        printk(KERN_INFO "Reached end of the device on a write");
    while ((nbytes < bytes_to_do) && !get_user(*tmp, (buf + nbytes))) {
        nbytes++;
        tmp++;
```

```
    }
    *ppos += nbytes;
    printk(KERN_INFO "\n Leaving the   WRITE function, nbytes=%d, pos=%d\n",
           nbytes, (int)*ppos);
    return nbytes;
}
```

To copy the value from the kernel space to a variable in user space, one can use:

```
#include <linux/uaccess.h>

int put_user(expr, ptr);
```

which copy value pointed by ptr at kernel space to the user space variable expr. Returns **0** for success, **-EFAULT** otherwise.

To copy a chunk of memory across space, one would use:

```
#include <linux/uaccess.h>

unsigned long copy_to_user (void __user * to, const void * from, unsigned long n);
unsigned long copy_from_user (void * to, const void __user * from, unsigned long n);
```

copy_from_user() copies data of size len from the user space memory pointed to by **from** to the kernel space memory pointed to by **to**. This function returns the number of bytes not transferred in error; otherwise returns 0.

copy_to_user() copies data from the kernel space to the user space.

The caller must check the return value for error, and if there are data not transferred, the caller needs return with **-EFAULT**, as illustrated by the following example.

In this example ioctl() determines the size of the data, then use copy_from_user() and copy_to_user() to transfer data across the user space and the kernel space.

```
#include <linux/uaccess.h>
#include <linux/module.h>

#define MYIOC_TYPE 'k'

static inline long
mycdrv_unlocked_ioctl(struct file *fp, unsigned int cmd, unsigned long arg)
{
    int i, rc, direction;
    int size;
    char *buffer;
    void __user *ioargp = (void __user *)arg;
    int copied = 0;

    /* make sure it is a valid command */

    if (_IOC_TYPE(cmd) != MYIOC_TYPE) {
        printk(KERN_WARNING " got invalid case, CMD=%d\n", cmd);
        return -EINVAL;
    }

    /* get the size of the buffer and kmalloc it */

    size = _IOC_SIZE(cmd);
    buffer = kmalloc((size_t) size, GFP_KERNEL);
    if (!buffer) {
        printk(KERN_ERR "Kmalloc failed for buffer\n");
        return -ENOMEM;
    }
```

```
    /* fill it with X */

    memset(buffer, 'X', size);

    direction = _IOC_DIR(cmd);

    switch (direction) {

    case _IOC_WRITE:
        printk
            (KERN_INFO
             " reading = %d bytes from user-space and writing to device\n",
             size);
        if(copy_from_user(&buffer[copied], &ioargp[copied], size))
                return -EFAULT;
        break;

    case _IOC_READ:
        printk(KERN_INFO
               " reading device and writing = %d bytes to user-space\n",
               size);
        if(copy_to_user(&ioargp[copied], &buffer[copied], size))
                return -EFAULT;
        break;

    default:
        printk(KERN_WARNING " got invalid case, CMD=%d\n", cmd);
        return -EINVAL;
    }
    for (i = 0; i < size; i++)
        printk(KERN_INFO "%c", buffer[i]);
    printk(KERN_INFO "\n");

    if (buffer)
        kfree(buffer);
    return rc;
}
```

1.6 Memory mapping

Memory copying has overheads, which can be eliminated by memory mapping from one space to the other, so that memory can be accessed by the other side directly without the expenses of copying.

When a file is memory mapped, the file can be associated with a range of linear addresses. Input and output operations on the file can be accomplished with simple memory references, rather than explicit I/O operations.

Memory mapping can also be done on device nodes for direct access to hardware devices. In this case, the driver must register and implement a proper mmap() entry point.

This method is not useful for stream-oriented devices. The mapped area must be a multiple of PAGE_SIZE extent, and start on a page boundary.

Two basic kinds of memory mapping exist:

Shared mapping - operations on the memory region is equivalent to changing to the file it represents. Changes are immediately visible to processes accessing the file.

Private mapping - operations on the memory region are not committed to the disk and invisible to other processes accessing the file. More efficient, but is designed to be used in read-only situations (e.g. final saving of data is done by writing to another file).

From the user side, memory mapping is done with:

```c
#include <unistd.h>
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);

int munmap(void *start, size_t length);
```

This requests the mapping into memory of length bytes, starting at offset offset, from the file specified by fd. The offset must be an integral number of pages.

The address start is a preferred address to map to. If **0** is given (the usual case), mmap() will choose the address and put it in the return value.

prot is the desired memory protection. It has bits:

PROT_EXEC	Page may be executed
PROT_READ	Page may be read
PROT_WRITE	Page may be written
PROT_NONE	Page may not be accessed

Except PROT_NONE, the above flags can be OR'ed.

flag specifies the type of mapped object. It has bits:

MAP_FIXED	If start can't be used, fail.
MAP_SHARED	Share the mapping with all other processes.
MAP_PRIVATE	Create a private copy-on-write mapping.
MAP_ANONYMOUS	Create a mapping only in memory, without a file association

Either MAP_SHARED or MAP_PRIVATE must be specified. Remember, a private mapping does not change the file on disk. Therefore any changes to it will be lost when the process terminates.

MAP_ANONYMOUS is a common way to share memory between the parent process and children processes. Here is an example:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
  int fd = -1;
  int size = 4096;
  int status;
  char *area;
  pid_t pid;

  area = mmap(NULL, size, PROT_READ | PROT_WRITE,
              MAP_SHARED | MAP_ANONYMOUS, fd, 0);

  pid = fork();

  if (0 == pid) {
    /* child */
    strcpy(area, "This is a message from the child");
    printf("Child has written: %s\n", area);
```

```
    exit(EXIT_SUCCESS);
  } else if (pid > 0){
    /* parent */
    wait(&status);  /* wait until child terminates */
    printf("Parent has read: %s\n", area);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

munmap() deletes the mapping and causes further references to addresses within the range to generate invalid memory references.

For the kernel side, the driver entry point looks like:

```
#include <linux/mm.h>

int (*mmap)(struct file *filp, struct vm_area_struct *vma);
```

The **vm_area_struct** data structure is defined in /usr/src/linux/include/linux/mm.h and contains the important information. The basic elements are:

```
struct vm_area_struct {
...
  unsigned long vm_start;  /* Our start address within vm_mm. */
  unsigned long vm_end;    /* The first byte after our end address within vm_mm. */
...
  pgprot_t vm_page_prot;   /* Access permissions of this VMA. */
  unsigned long vm_flags;  /* Flags, listed below */
...
  /* Functions pointers to deal with this struct */
  struct vm_operations_struct *vm_ops;

  /* Information about our backing store: */
  unsigned long vm_pgoff;  /* Offset (within vm file) in PAGE_SIZE
                              units, *not* PAGE_CACHE_SIZE */
...
}
```

Like the fops structure discussed in lab 3, the vm_ops structure can be used to override default operations. Pointers can be given for functions to: open(), close(), protect(), sync(), advice(), swapout(), swapin()...

Here is a simple example to show how the fields are used:

```
#include <linux/mm.h>

int my_mmap(struct file *file, struct vm_area_struct *vma) {
  if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff,
                      vma->vm_end - vma->vm_start, vm->vm_page_prot))
    return -EAGAIN;
  return 0;
}
```

Most of the work is done by the function:

```
#include <linux/mm.h>

int remap_pfn_range(struct vm_area_struct *vma, unsigned long start_addr,
                    unsigned long pfn, unsigned long size, pgprot_t prot);
```

which maps pages to the specified vma range, where:

- **vma** points to the vma struct

- **start_addr** is the beginning of the mapping address
- **pfn** stands for "page frame number" of the page (or the first page, in case of mapping contiguous pages in one go)
- **size** is the size of the mapping, must be a multiple of page sizes
- **prot** is the protection setting of the page(s).

The page frame number of a page can be obtained by one of the functions:

```
unsigned long page_to_pfn(struct page *page);

unsigned long virt_to_pfn(void *addr);
```

For contiguous pages, remap_pfn_range() can map the entire mapping range requested in one go. However in cases where pages are not contiguous, as in the assignment one, you need to map each page one by one.

Note that this function does allow mapping memory above the 4GB barrier.

Here is a simple example of a program to test the mmap() entry:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <errno.h>

#define DEATH(mess) { perror(mess); exit(errno); }

int main(int argc, char **argv) {
    int fd, size, rc, j;
    char *area, tmp, *nodename = "/dev/mycdrv";
    char c[2] = "CX";

    if (argc > 1)
        nodename = argv[1];

    size = getpagesize();   /* use one page by default */

    if (argc > 2)
        size = atoi(argv[2]);

    printf("Memory Mapping Node: %s, of size %d bytes\n", nodename, size);

    if ((fd = open (nodename, O_RDWR)) < 0)
        DEATH ("problems opening the node ");

    area = mmap (NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    if (area == MAP_FAILED)
        DEATH ("error mmaping");

    /* can close the file now */

    close (fd);

    /* put the string repeatedly in the file */

    tmp = area;
    for (j = 0; j < size - 1; j += 2, tmp += 2)
        memcpy (tmp, &c, 2);
```

```
    /* just cat out the file to see if it worked */

    rc = write (STDOUT_FILENO, area, size);

    if (rc != size)
        DEATH ("problems writing");

    exit (EXIT_SUCCESS);
}
```

Here is a simple driver with a mmap() entry point:

```
#include <linux/module.h>       /* for modules */
#include <linux/fs.h>           /* file_operations */
#include <linux/uaccess.h>      /* copy_(to,from)_user */
#include <linux/init.h>         /* module_init, module_exit */
#include <linux/slab.h>         /* kmalloc */
#include <linux/cdev.h>         /* cdev utilities */
#include <linux/mm.h>           /* remap_pfn_range */

#define MYDEV_NAME "mycdrv"

static dev_t first;
static unsigned int count = 1;
static int my_major = 700, my_minor = 0;
static struct cdev *my_cdev;

static int mycdrv_mmap (struct file *file, struct vm_area_struct *vma)
{
    printk (KERN_INFO "I entered the mmap function\n");
    if (remap_pfn_range (vma, vma->vm_start,
                         vma->vm_pgoff,
                         vma->vm_end - vma->vm_start, vma->vm_page_prot)) {
        return -EAGAIN;
    }

    return 0;
}

/* don't bother with open, release, read and write */

static struct file_operations mycdrv_fops = {
    .owner = THIS_MODULE,
    .mmap = mycdrv_mmap,
};
static int __init my_init (void)
{
    first = MKDEV (my_major, my_minor);
    register_chrdev_region (first, count, MYDEV_NAME);
    my_cdev = cdev_alloc ();
    cdev_init (my_cdev, &mycdrv_fops);
    cdev_add (my_cdev, first, count);
    printk (KERN_INFO "\nSucceeded in registering character device %s\n",
            MYDEV_NAME);
    return 0;
}
static void __exit my_exit (void)
{
    cdev_del (my_cdev);
    unregister_chrdev_region (first, count);
    printk (KERN_INFO "\ndevice unregistered\n");
```

```
}

module_init (my_init);
module_exit (my_exit);

MODULE_AUTHOR ("Jerry Cooperstein");
MODULE_DESCRIPTION ("LDD:1.0 s_18/mmapdrv.c");
MODULE_LICENSE ("GPL v2");
```

1.7 Atomic operations

In multithreaded architectures, there can be more than one process accessing (making a system call) on the same device concurrently. Therefore care must be taken to protect access of the shared data, or data race can occur if more than one processes write to the same location simultaneously.

For integer-sized data, the Linux kernel API provides the atomic type **atomic_t**, which is accessed by atomic functions as one single instruction. **atomic_t** is defined as:

```
typedef struct {
    volatile int counter;
} atomic_t;
```

Atomic variables with the type atomic_t must be accessed by the following functions:

```
atomic_read(atomic_t *v);
atomic_set(atomic_t *v, int i);

void atomic_add (int i, atomic_t *v);
void atomic_sub (int i, atomic_t *v);
void atomic_inc (atomic_t *v);
void atomic_dec (atomic_t *v);

int atomic_dec_and_test (atomic_t *v);
int atomic_inc_and_test_greater_zero (atomic_t *v);
int atomic_sub_and_test (int i, atomic_t *v);
int atomic_add_negative (int i, atomic_t *v);
int atomic_sub_return (int i, atomic_t *v);
int atomic_add_return (int i, atomic_t *v);
int atomic_inc_return (int i, atomic_t *v);
int atomic_dec_return (int i, atomic_t *v);
```

Now go back to assignment 1 and add a mmap() entry point that maps the ramdisk to user space. as in the assignment specification.