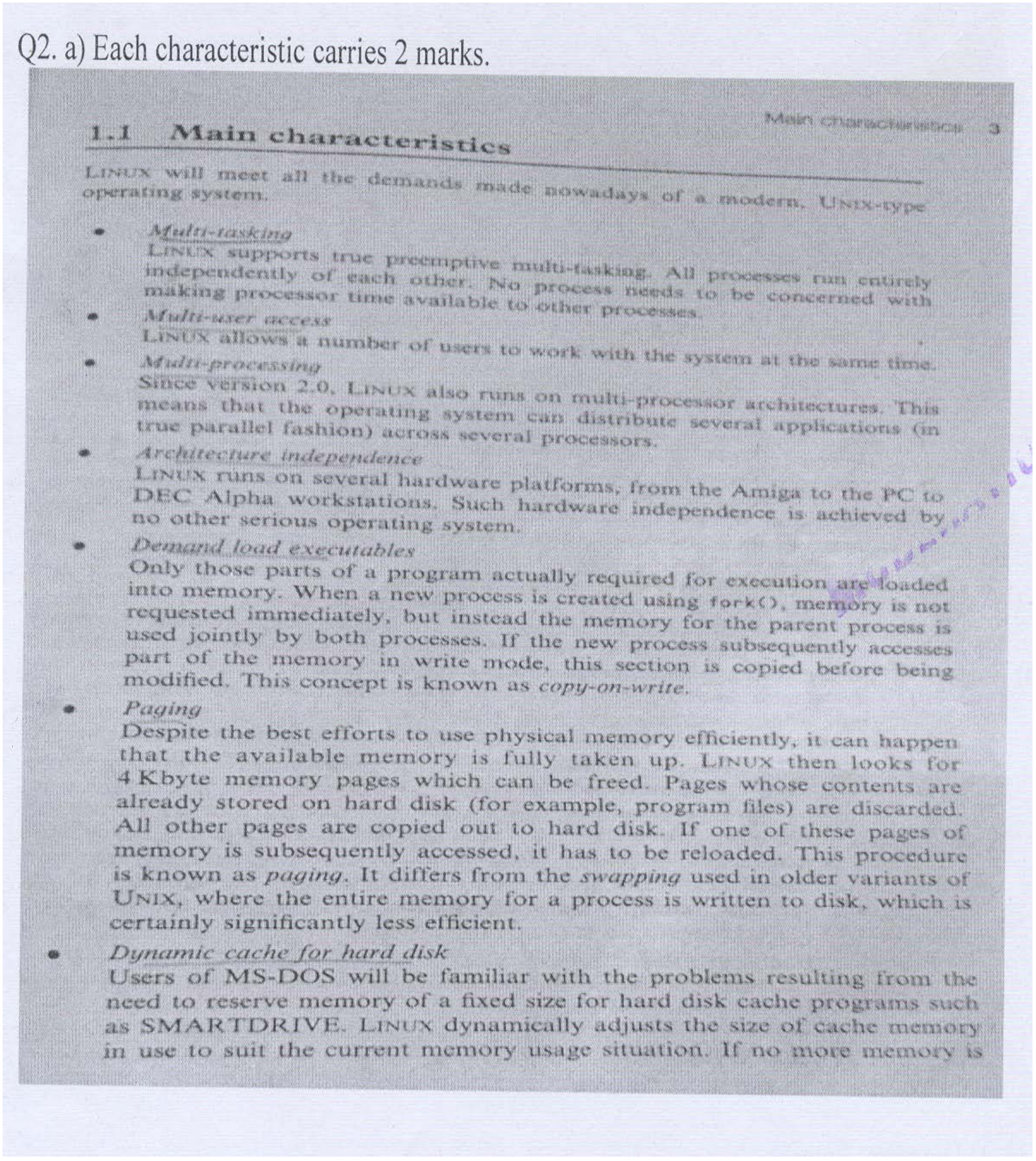
Q.2a. Explain any 5 main characteristics of LINUX.

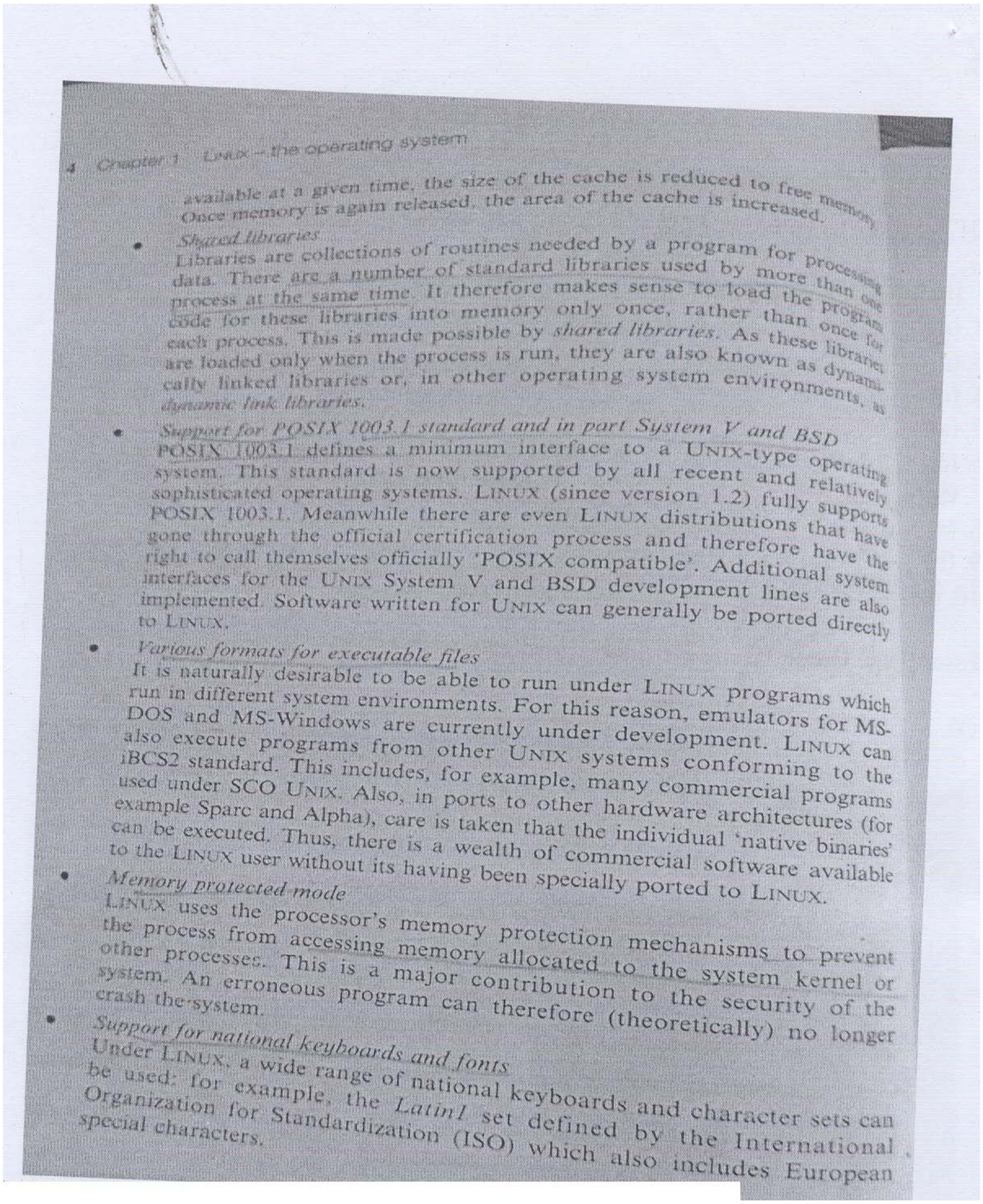


Q2. a) Each characteristic carries 2 marks.

1.1 Main characteristics

LINUX will meet all the demands made nowadays of a modern, UNIX-type operating system.

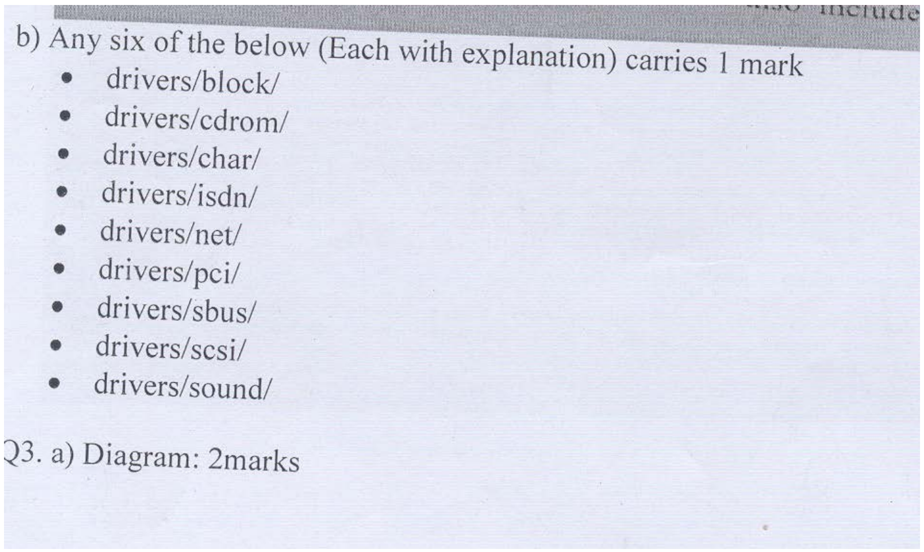
- *Multi-tasking*
  LINUX supports true preemptive multi-tasking. All processes run entirely independently of each other. No process needs to be concerned with making processor time available to other processes.
- *Multi-user access*
  LINUX allows a number of users to work with the system at the same time.
- *Multi-processing*
  Since version 2.0, LINUX also runs on multi-processor architectures. This means that the operating system can distribute several applications (in true parallel fashion) across several processors.
- *Architecture independence*
  LINUX runs on several hardware platforms, from the Amiga to the PC to DEC Alpha workstations. Such hardware independence is achieved by no other serious operating system.
- *Demand load executables*
  Only those parts of a program actually required for execution are loaded into memory. When a new process is created using `fork()`, memory is not requested immediately, but instead the memory for the parent process is used jointly by both processes. If the new process subsequently accesses part of the memory in write mode, this section is copied before being modified. This concept is known as *copy-on-write*.
- *Paging*
  Despite the best efforts to use physical memory efficiently, it can happen that the available memory is fully taken up. LINUX then looks for 4 Kbyte memory pages which can be freed. Pages whose contents are already stored on hard disk (for example, program files) are discarded. All other pages are copied out to hard disk. If one of these pages of memory is subsequently accessed, it has to be reloaded. This procedure is known as *paging*. It differs from the *swapping* used in older variants of UNIX, where the entire memory for a process is written to disk, which is certainly significantly less efficient.
- *Dynamic cache for hard disk*
  Users of MS-DOS will be familiar with the problems resulting from the need to reserve memory of a fixed size for hard disk cache programs such as SMARTDRIVE. LINUX dynamically adjusts the size of cache memory in use to suit the current memory usage situation. If no more memory is



available at a given time, the size of the cache is reduced to free memory. Once memory is again released, the area of the cache is increased.

- *Shared libraries*
  Libraries are collections of routines needed by a program for processing data. There are a number of standard libraries used by more than one process at the same time. It therefore makes sense to load the program code for these libraries into memory only once, rather than once for each process. This is made possible by *shared libraries*. As these libraries are loaded only when the process is run, they are also known as dynamically linked libraries or, in other operating system environments, as *dynamic link libraries*.
- *Support for POSIX 1003.1 standard and in part System V and BSD*
  POSIX 1003.1 defines a minimum interface to a UNIX-type operating system. This standard is now supported by all recent and relatively sophisticated operating systems. LINUX (since version 1.2) fully supports POSIX 1003.1. Meanwhile there are even LINUX distributions that have gone through the official certification process and therefore have the right to call themselves officially 'POSIX compatible'. Additional system interfaces for the UNIX System V and BSD development lines are also implemented. Software written for UNIX can generally be ported directly to LINUX.
- *Various formats for executable files*
  It is naturally desirable to be able to run under LINUX programs which run in different system environments. For this reason, emulators for MS-DOS and MS-Windows are currently under development. LINUX can also execute programs from other UNIX systems conforming to the iBCS2 standard. This includes, for example, many commercial programs used under SCO UNIX. Also, in ports to other hardware architectures (for example Sparc and Alpha), care is taken that the individual 'native binaries' can be executed. Thus, there is a wealth of commercial software available to the LINUX user without its having been specially ported to LINUX.
- *Memory protected mode*
  LINUX uses the processor's memory protection mechanisms to prevent the process from accessing memory allocated to the system kernel or other processes. This is a major contribution to the security of the system. An erroneous program can therefore (theoretically) no longer crash the system.
- *Support for national keyboards and fonts*
  Under LINUX, a wide range of national keyboards and character sets can be used: for example, the *Latin1* set defined by the International Organization for Standardization (ISO) which also includes European special characters.

b.Classify drivers/ directory according to their subdirectories.



b) Any six of the below (Each with explanation) carries 1 mark

- drivers/block/
- drivers/cdrom/
- drivers/char/
- drivers/isdn/
- drivers/net/
- drivers/pci/
- drivers/sbus/
- drivers/scsi/
- drivers/sound/

Q3. a) Diagram: 2marks

Q.3a. Describe the states within a process with the help of a neat diagram.

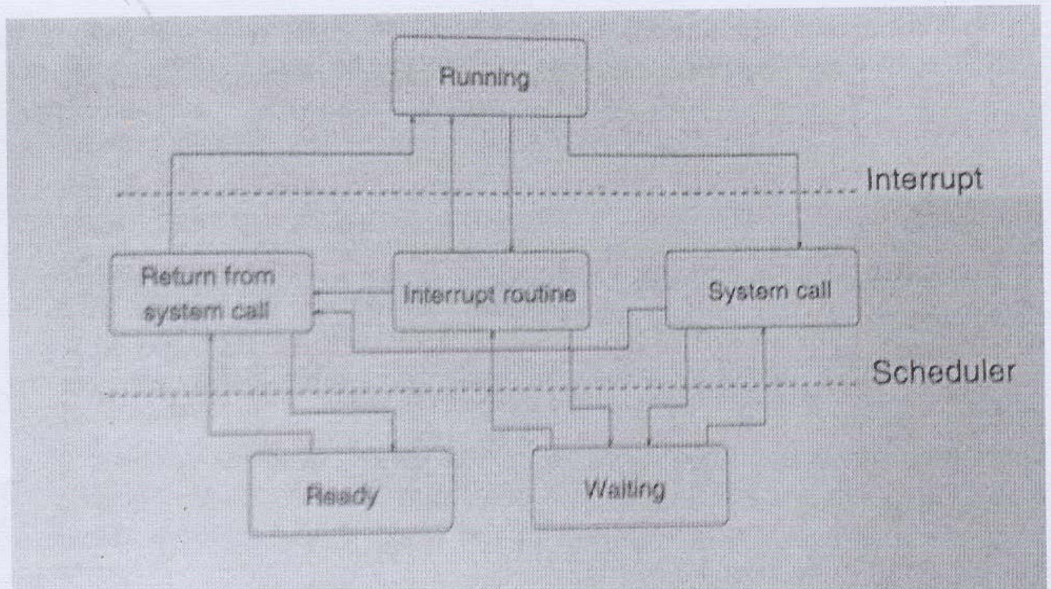
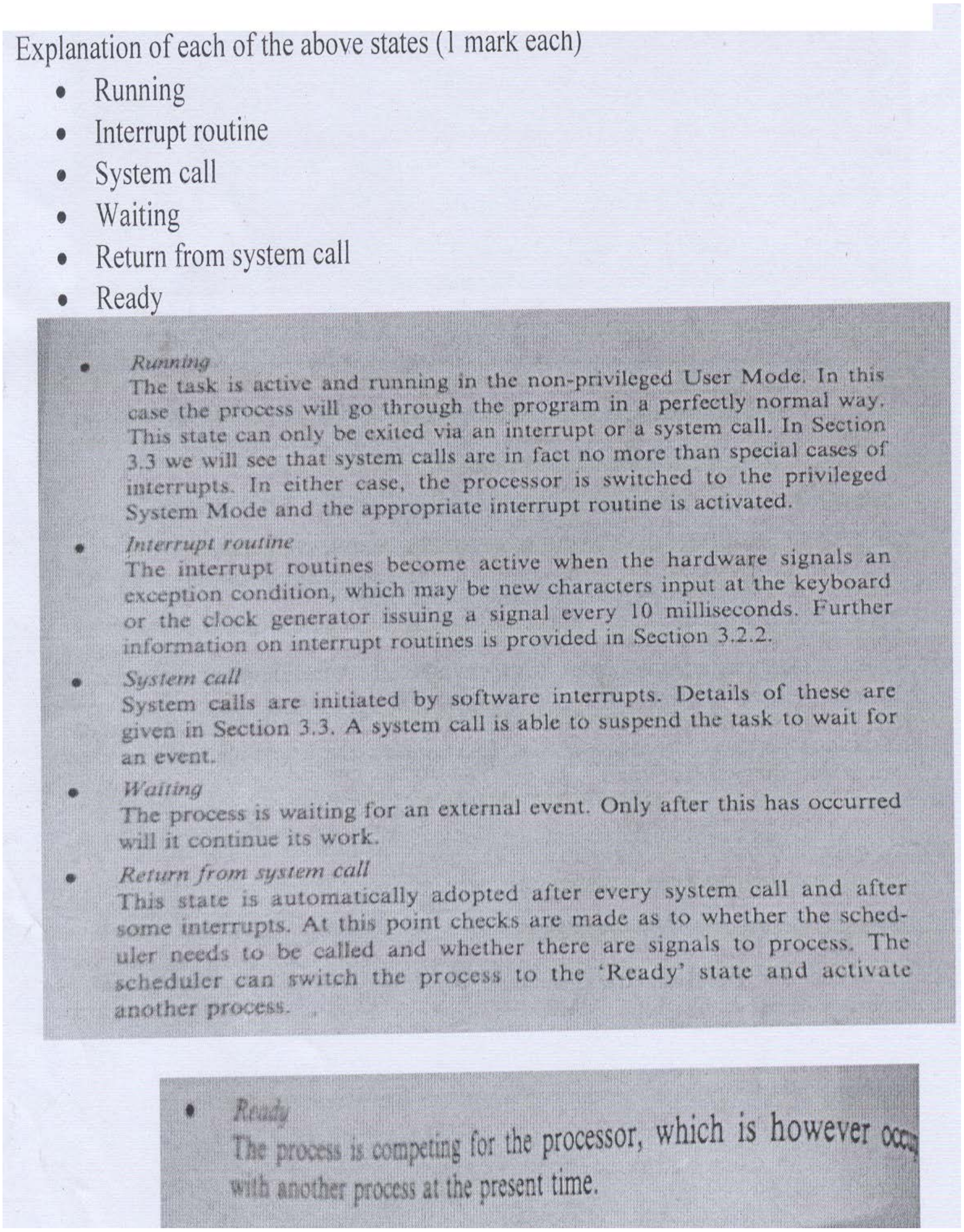


Figure 3.2 Chart of states within a process.



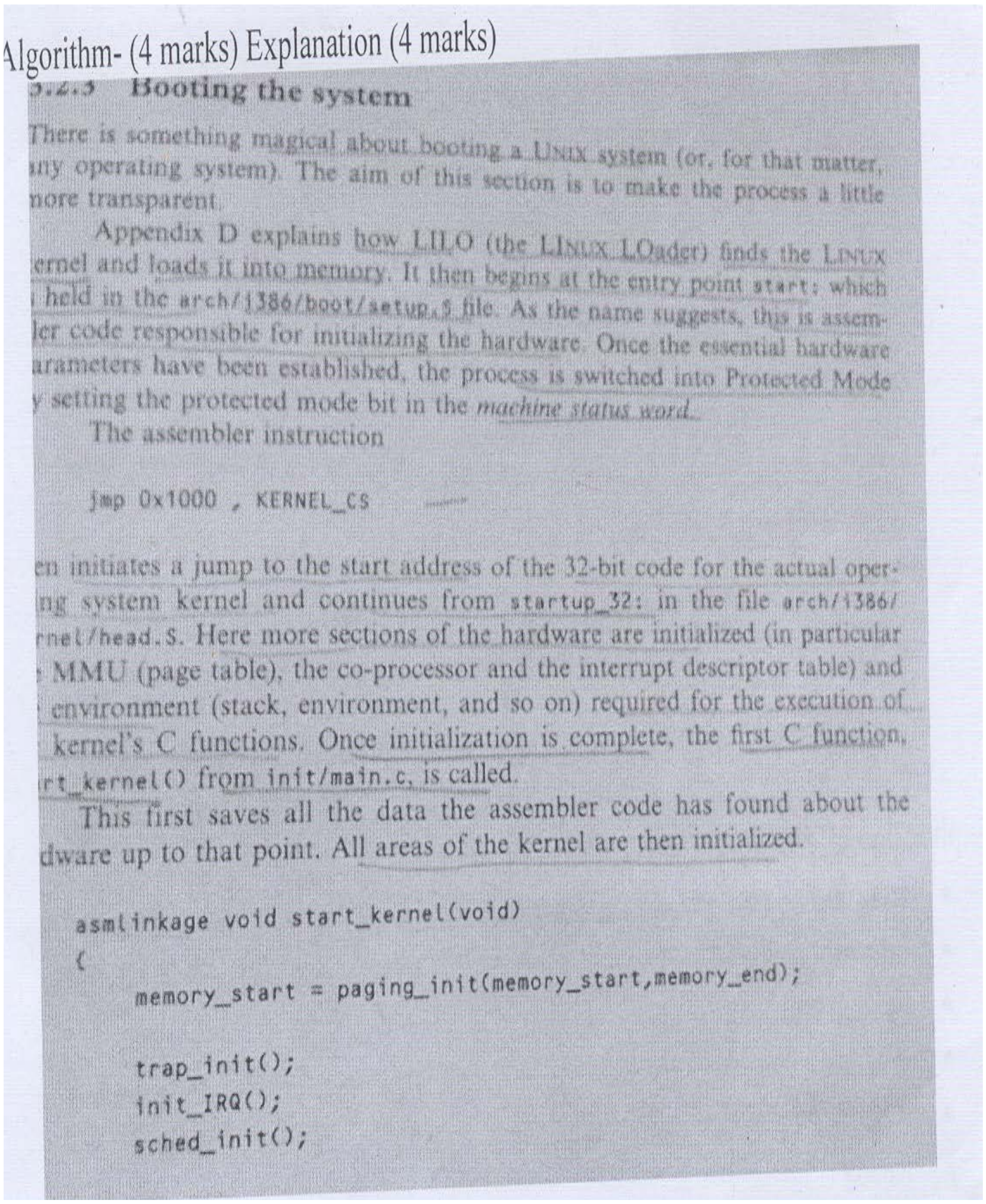
Explanation of each of the above states (1 mark each)

- Running
- Interrupt routine
- System call
- Waiting
- Return from system call
- Ready

- *Running*
  The task is active and running in the non-privileged User Mode. In this case the process will go through the program in a perfectly normal way. This state can only be exited via an interrupt or a system call. In Section 3.3 we will see that system calls are in fact no more than special cases of interrupts. In either case, the processor is switched to the privileged System Mode and the appropriate interrupt routine is activated.
- *Interrupt routine*
  The interrupt routines become active when the hardware signals an exception condition, which may be new characters input at the keyboard or the clock generator issuing a signal every 10 milliseconds. Further information on interrupt routines is provided in Section 3.2.2.
- *System call*
  System calls are initiated by software interrupts. Details of these are given in Section 3.3. A system call is able to suspend the task to wait for an event.
- *Waiting*
  The process is waiting for an external event. Only after this has occurred will it continue its work.
- *Return from system call*
  This state is automatically adopted after every system call and after some interrupts. At this point checks are made as to whether the scheduler needs to be called and whether there are signals to process. The scheduler can switch the process to the 'Ready' state and activate another process.

- *Ready*
  The process is competing for the processor, which is however occu
  with another process at the present time.

b.Write and explain the algorithm for the booting of a LINUX system.



Algorithm- (4 marks) Explanation (4 marks)

5.2.3 Booting the system

There is something magical about booting a LINUX system (or, for that matter, any operating system). The aim of this section is to make the process a little more transparent.

Appendix D explains how LILO (the LINUX LOader) finds the LINUX ernel and loads it into memory. It then begins at the entry point `start:` which held in the `arch/i386/boot/setup.S` file. As the name suggests, this is assemler code responsible for initializing the hardware. Once the essential hardware arameters have been established, the process is switched into Protected Mode y setting the protected mode bit in the *machine status word*.

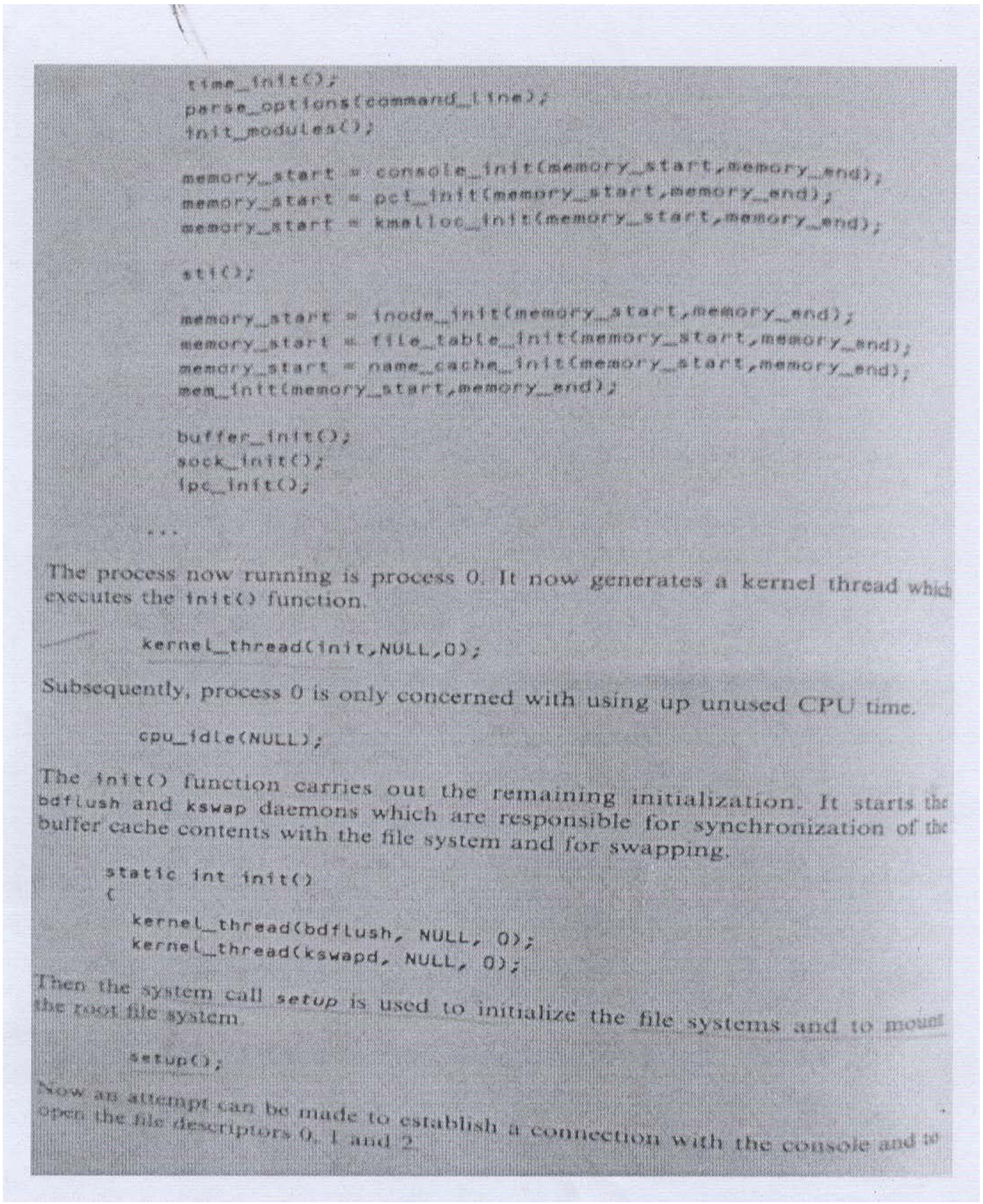
The assembler instruction

```
jmp 0x1000 , KERNEL_CS
```

en initiates a jump to the start address of the 32-bit code for the actual operng system kernel and continues from `startup_32:` in the file `arch/i386/ rnel/head.S`. Here more sections of the hardware are initialized (in particular MMU (page table), the co-processor and the interrupt descriptor table) and environment (stack, environment, and so on) required for the execution of kernel's C functions. Once initialization is complete, the first C function, `rt_kernel()` from `init/main.c`, is called.

This first saves all the data the assembler code has found about the dware up to that point. All areas of the kernel are then initialized.

```
asmlinkage void start_kernel(void)
{
        memory_start = paging_init(memory_start,memory_end);

        trap_init();
        init_IRQ();
        sched_init();
```



```
        time_init();
        parse_options(command_line);
        init_modules();

        memory_start = console_init(memory_start,memory_end);
        memory_start = pci_init(memory_start,memory_end);
        memory_start = kmalloc_init(memory_start,memory_end);

        sti();

        memory_start = inode_init(memory_start,memory_end);
        memory_start = file_table_init(memory_start,memory_end);
        memory_start = name_cache_init(memory_start,memory_end);
        mem_init(memory_start,memory_end);

        buffer_init();
        sock_init();
        ipc_init();

        ...
```

The process now running is process 0. It now generates a kernel thread which executes the `init()` function.

```
        kernel_thread(init,NULL,0);
```

Subsequently, process 0 is only concerned with using up unused CPU time.

```
        cpu_idle(NULL);
```

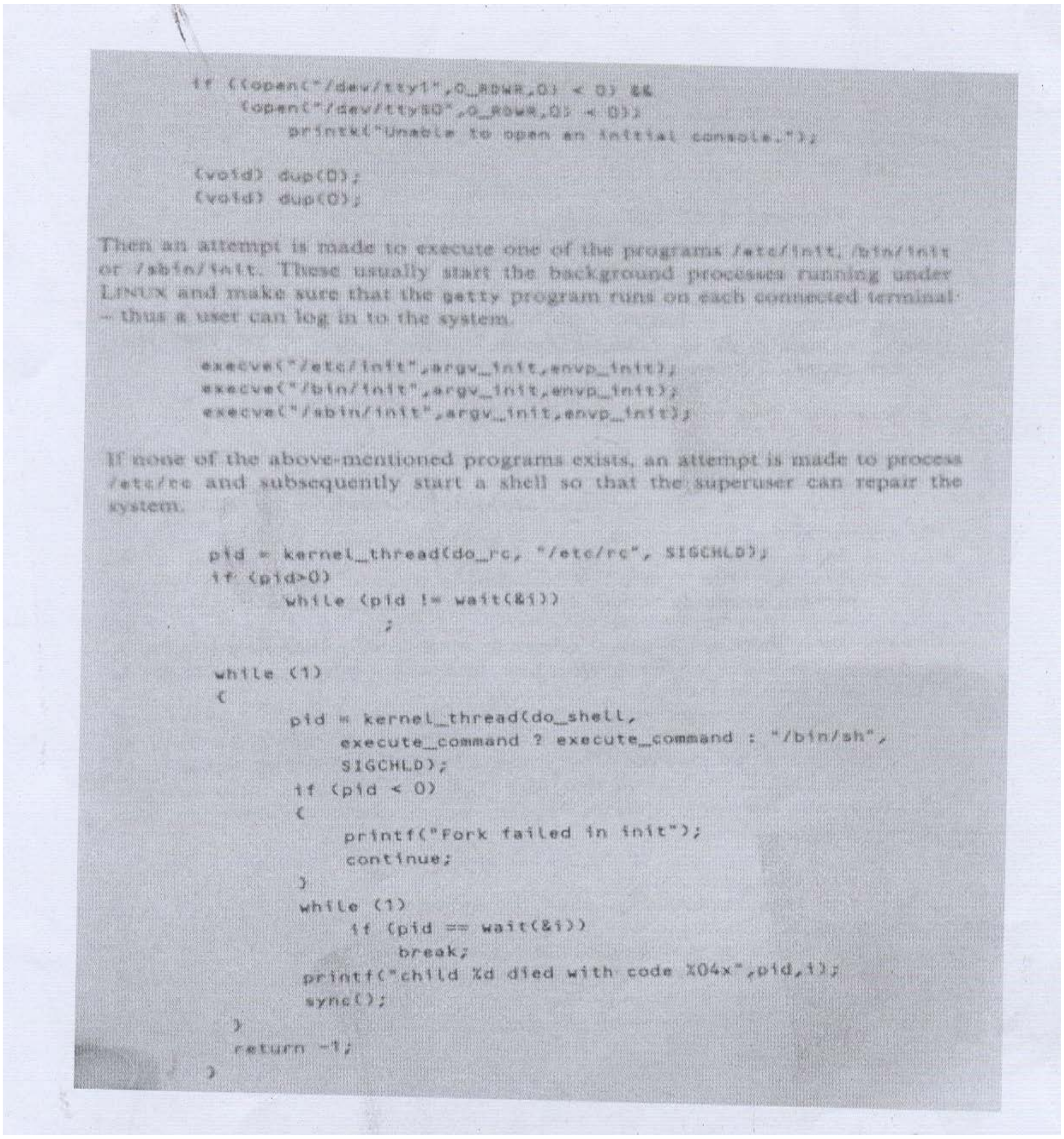
The `init()` function carries out the remaining initialization. It starts the `bdflush` and `kswap` daemons which are responsible for synchronization of the buffer cache contents with the file system and for swapping.

```
    static int init()
    {
        kernel_thread(bdflush, NULL, 0);
        kernel_thread(kswapd, NULL, 0);
```

Then the system call *setup* is used to initialize the file systems and to mount the root file system.

```
        setup();
```

Now an attempt can be made to establish a connection with the console and to open the file descriptors 0, 1 and 2.



```
        if ((open("/dev/tty1",O_RDWR,0) < 0) &&
            (open("/dev/ttyS0",O_RDWR,0) < 0))
                printk("Unable to open an initial console.");

        (void) dup(0);
        (void) dup(0);
```

Then an attempt is made to execute one of the programs /etc/init, /bin/init or /sbin/init. These usually start the background processes running under LINUX and make sure that the getty program runs on each connected terminal – thus a user can log in to the system.

```
        execve("/etc/init",argv_init,envp_init);
        execve("/bin/init",argv_init,envp_init);
        execve("/sbin/init",argv_init,envp_init);
```

If none of the above-mentioned programs exists, an attempt is made to process /etc/rc and subsequently start a shell so that the superuser can repair the system.

```
        pid = kernel_thread(do_rc, "/etc/rc", SIGCHLD);
        if (pid>0)
                while (pid != wait(&i))
                        ;

        while (1)
        {
                pid = kernel_thread(do_shell,
                    execute_command ? execute_command : "/bin/sh",
                    SIGCHLD);
                if (pid < 0)
                {
                    printf("Fork failed in init");
                    continue;
                }
                while (1)
                    if (pid == wait(&i))
                        break;
                printf("child %d died with code %04x",pid,i);
                sync();
        }
        return -1;
}
```

Q.4a. Explain with a neat diagram, the Linear address conversion in the architecture-independent memory model.

Q4.a) Diagram (3 marks) Explanation (3 marks)

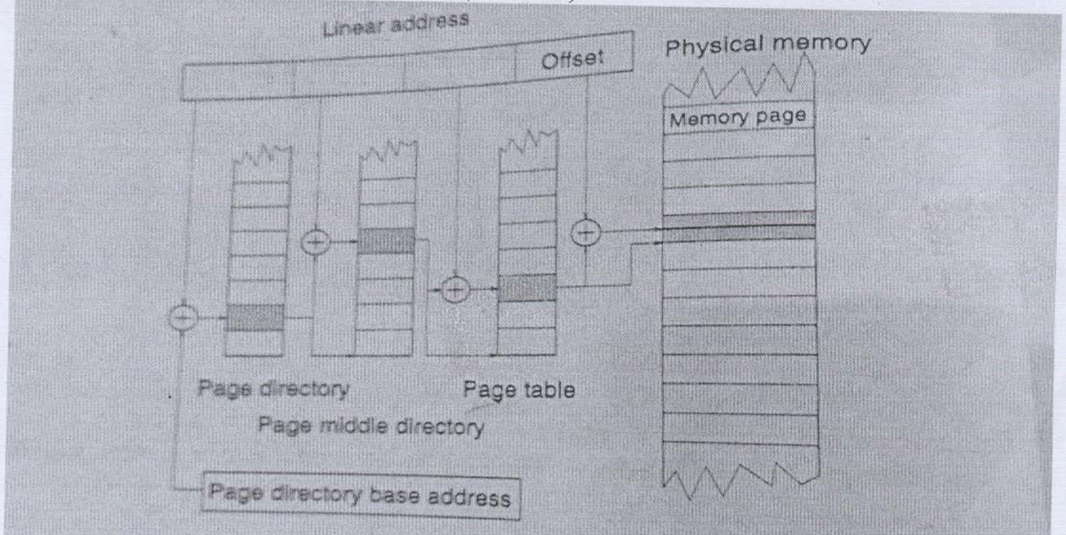
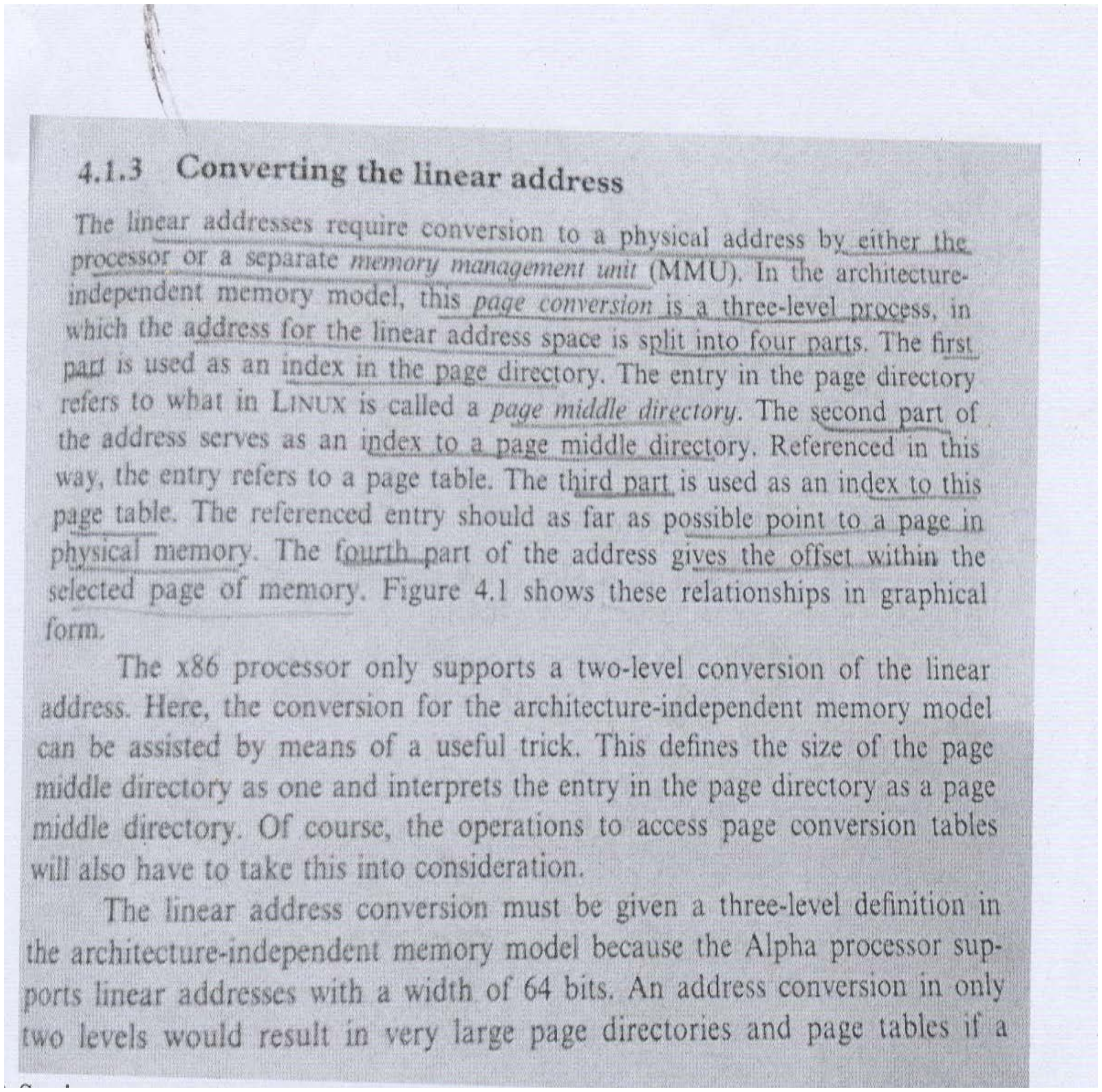


Figure 4.1 Linear address conversion in the architecture-independent memory model

6



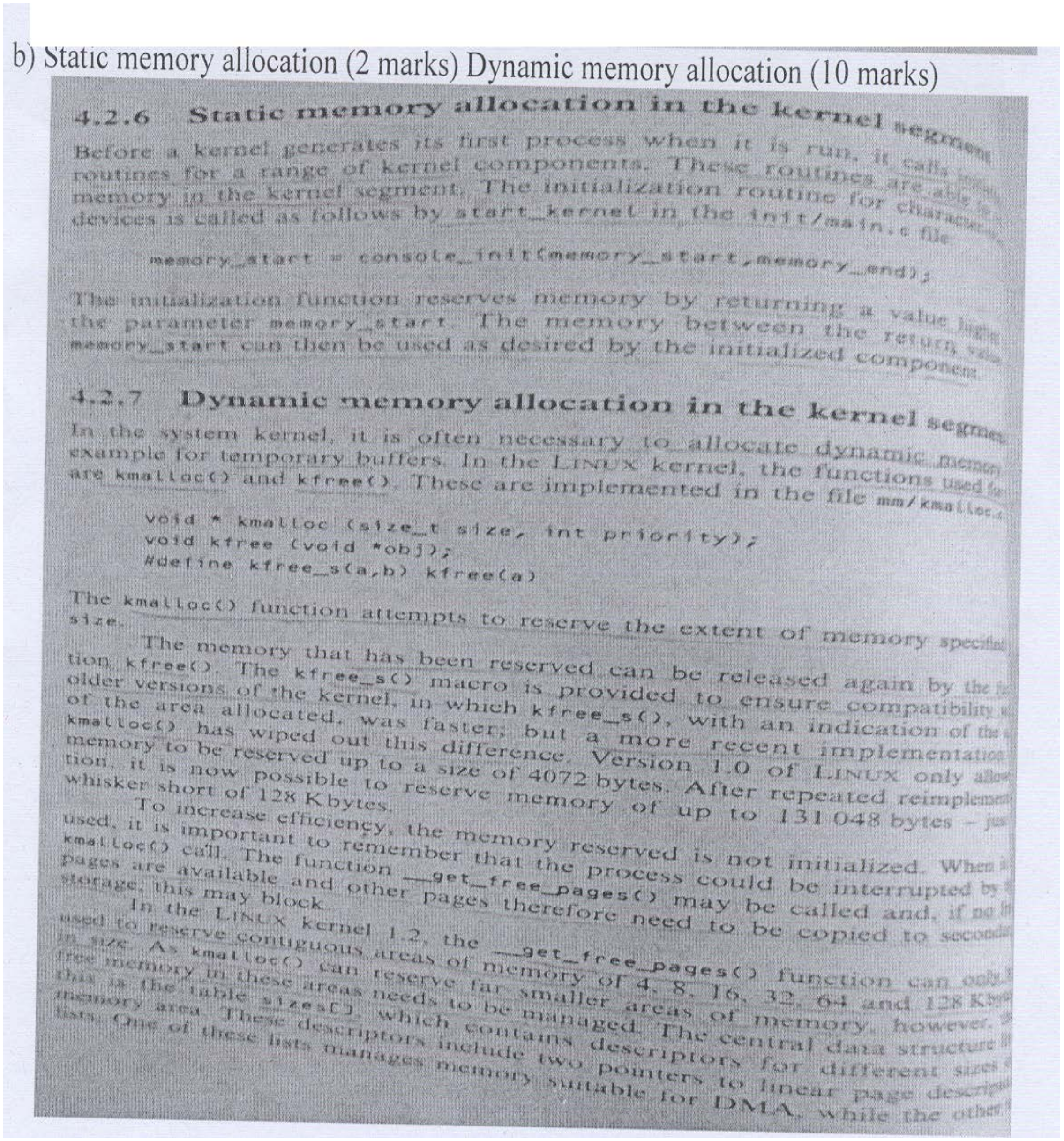
4.1.3 Converting the linear address

The linear addresses require conversion to a physical address by either the processor or a separate *memory management unit* (MMU). In the architecture-independent memory model, this *page conversion* is a three-level process, in which the address for the linear address space is split into four parts. The first part is used as an index in the page directory. The entry in the page directory refers to what in LINUX is called a *page middle directory*. The second part of the address serves as an index to a page middle directory. Referenced in this way, the entry refers to a page table. The third part is used as an index to this page table. The referenced entry should as far as possible point to a page in physical memory. The fourth part of the address gives the offset within the selected page of memory. Figure 4.1 shows these relationships in graphical form.

The x86 processor only supports a two-level conversion of the linear address. Here, the conversion for the architecture-independent memory model can be assisted by means of a useful trick. This defines the size of the page middle directory as one and interprets the entry in the page directory as a page middle directory. Of course, the operations to access page conversion tables will also have to take this into consideration.

The linear address conversion must be given a three-level definition in the architecture-independent memory model because the Alpha processor supports linear addresses with a width of 64 bits. An address conversion in only two levels would result in very large page directories and page tables if a

b.What do you understand by static and dynamic memory allocation in the kernel segment? Explain.



b) Static memory allocation (2 marks) Dynamic memory allocation (10 marks)

4.2.6 Static memory allocation in the kernel segmen

Before a kernel generates its first process when it is run, it calls routines for a range of kernel components. These routines are able to memory in the kernel segment. The initialization routine for character devices is called as follows by start_kernel in the init/main.c file

```
memory_start = console_init(memory_start,memory_end);
```

The initialization function reserves memory by returning a value high the parameter memory_start. The memory between the return val memory_start can then be used as desired by the initialized componen

4.2.7 Dynamic memory allocation in the kernel segme

In the system kernel, it is often necessary to allocate dynamic memor example for temporary buffers. In the LINUX kernel, the functions used fo are kmalloc() and kfree(). These are implemented in the file mm/kmalloc.

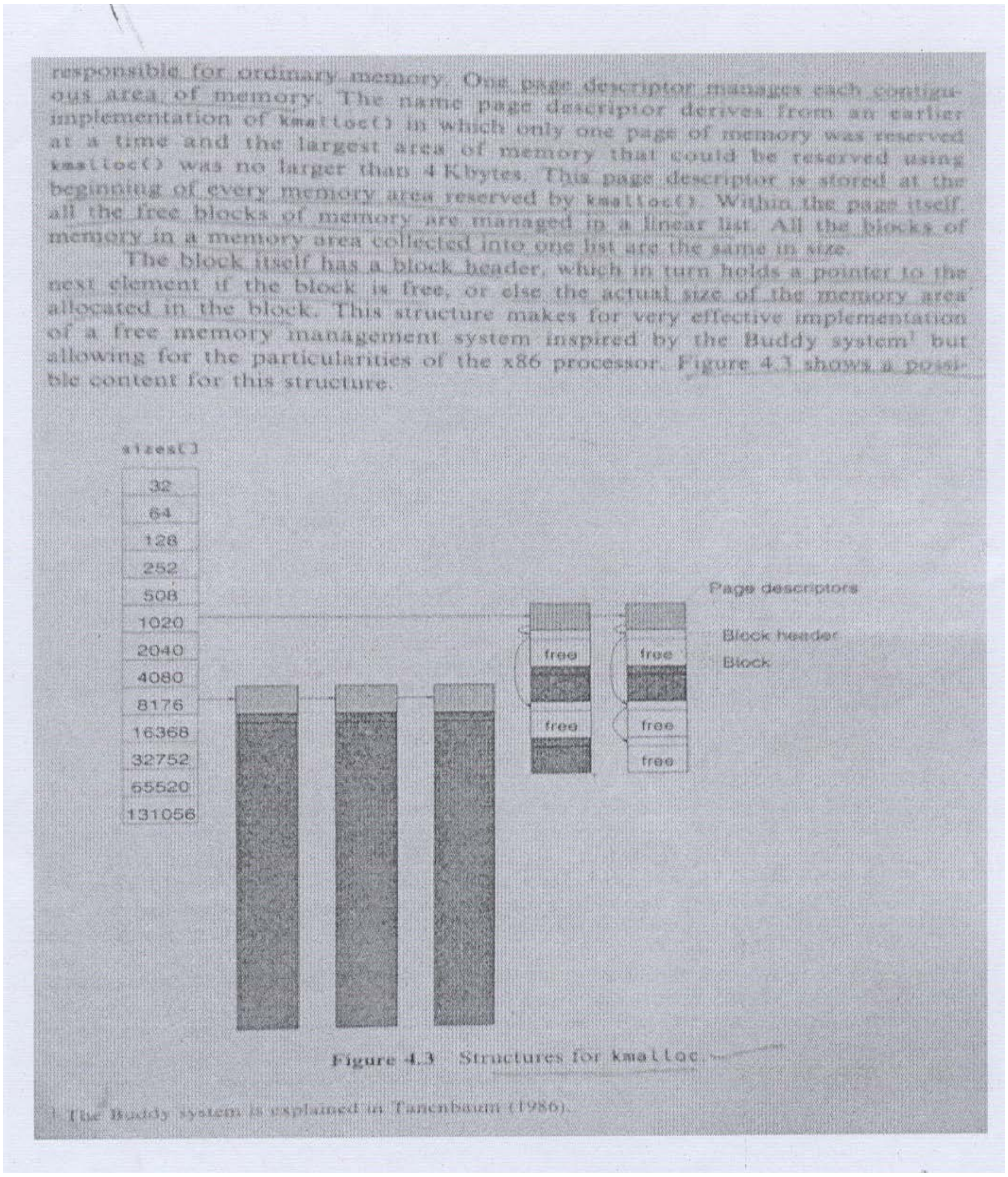
```
void * kmalloc (size_t size, int priority);
void kfree (void *obj);
#define kfree_s(a,b) kfree(a)
```

The kmalloc() function attempts to reserve the extent of memory specifie size.

The memory that has been reserved can be released again by the f tion kfree(). The kfree_s() macro is provided to ensure compatibility w older versions of the kernel, in which kfree_s(), with an indication of the of the area allocated, was faster; but a more recent implementation kmalloc() has wiped out this difference. Version 1.0 of LINUX only allow memory to be reserved up to a size of 4072 bytes. After repeated reimplemen tion, it is now possible to reserve memory of up to 131 048 bytes – jus whisker short of 128 Kbytes.

To increase efficiency, the memory reserved is not initialized. When i used, it is important to remember that the process could be interrupted by t kmalloc() call. The function __get_free_pages() may be called and, if no li pages are available and other pages therefore need to be copied to second storage, this may block.

In the LINUX kernel 1.2, the __get_free_pages() function can only b used to reserve contiguous areas of memory of 4, 8, 16, 32, 64 and 128 Kbyt in size. As kmalloc() can reserve far smaller areas of memory, however, th free memory in these areas needs to be managed. The central data structure fo this is the table sizes[], which contains descriptors for different sizes o memory area. These descriptors include two pointers to linear page descript lists. One of these lists manages memory suitable for DMA, while the other



responsible for ordinary memory. One page descriptor manages each contiguous area of memory. The name page descriptor derives from an earlier implementation of `kmalloc()` in which only one page of memory was reserved at a time and the largest area of memory that could be reserved using `kmalloc()` was no larger than 4 Kbytes. This page descriptor is stored at the beginning of every memory area reserved by `kmalloc()`. Within the page itself all the free blocks of memory are managed in a linear list. All the blocks of memory in a memory area collected into one list are the same in size.

The block itself has a block header, which in turn holds a pointer to the next element if the block is free, or else the actual size of the memory area allocated in the block. This structure makes for very effective implementation of a free memory management system inspired by the Buddy system[1] but allowing for the particularities of the x86 processor. Figure 4.3 shows a possible content for this structure.

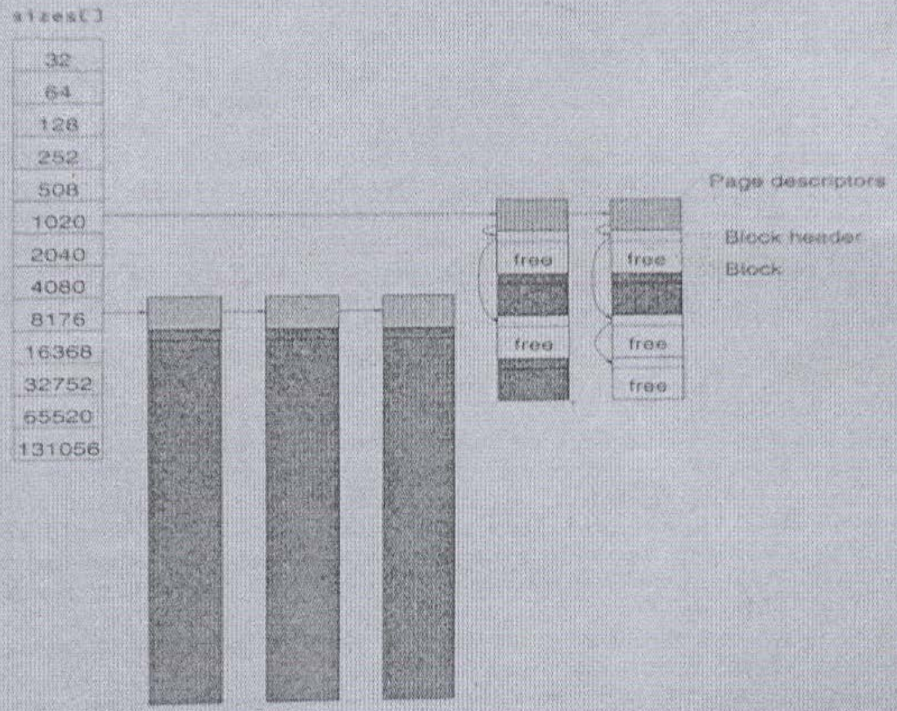
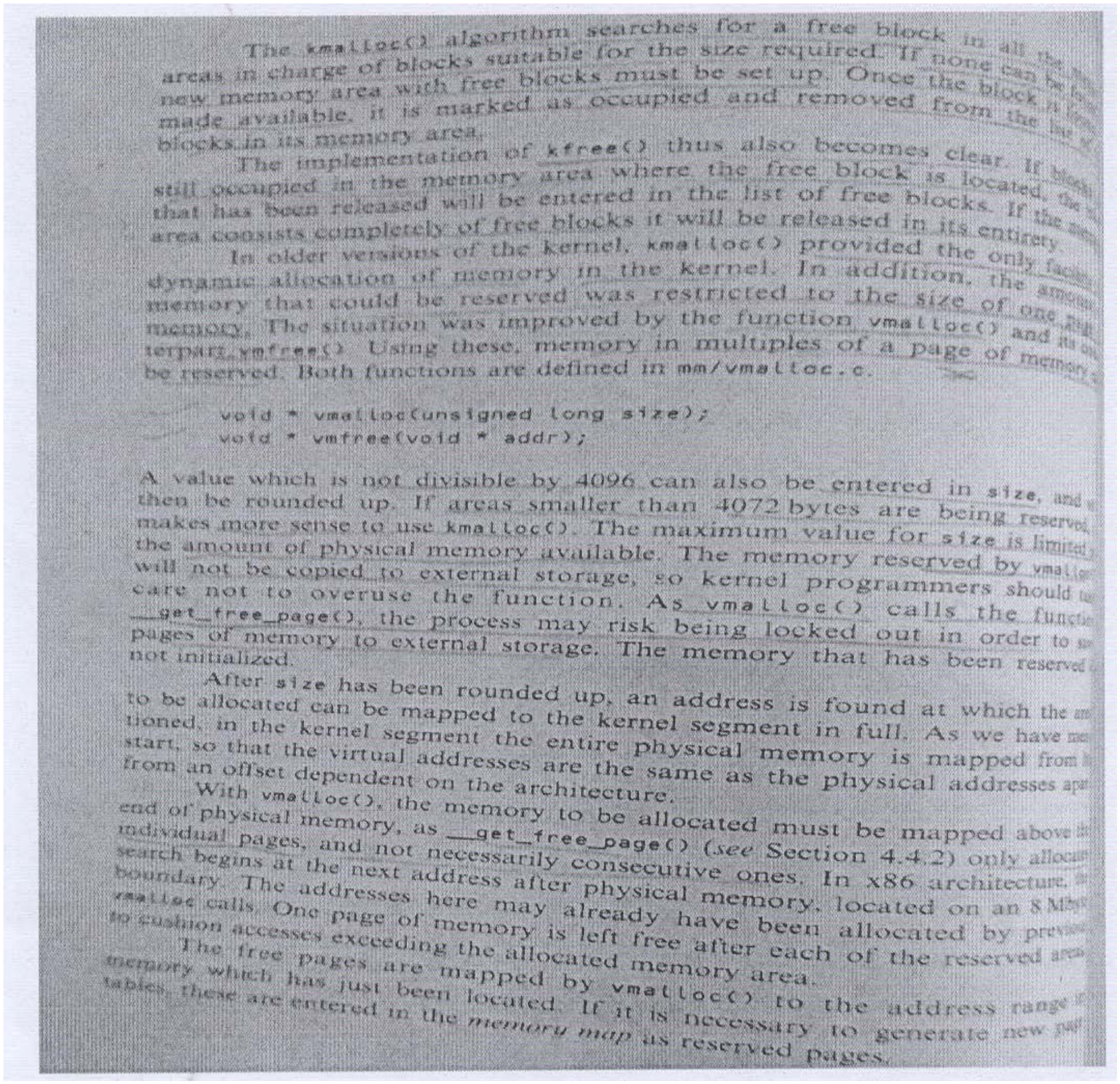


Figure 4.3 Structures for `kmalloc`.

[1] The Buddy system is explained in Tanenbaum (1986).



The kmalloc() algorithm searches for a free block in all the
areas in charge of blocks suitable for the size required. If none can be
new memory area with free blocks must be set up. Once the block is
made available, it is marked as occupied and removed from the list of
blocks in its memory area.

The implementation of kfree() thus also becomes clear. If blocks
still occupied in the memory area where the free block is located, the
that has been released will be entered in the list of free blocks. If the
area consists completely of free blocks it will be released in its entirety.

In older versions of the kernel, kmalloc() provided the only facility
dynamic allocation of memory in the kernel. In addition, the amount
memory that could be reserved was restricted to the size of one
memory. The situation was improved by the function vmalloc() and its
terpart vmfree(). Using these, memory in multiples of a page of memory
be reserved. Both functions are defined in mm/vmalloc.c.

```
void * vmalloc(unsigned long size);
void * vmfree(void * addr);
```

A value which is not divisible by 4096 can also be entered in size, and
then be rounded up. If areas smaller than 4072 bytes are being reserved,
makes more sense to use kmalloc(). The maximum value for size is limited
the amount of physical memory available. The memory reserved by vmalloc
will not be copied to external storage, so kernel programmers should
care not to overuse the function. As vmalloc() calls the functi
__get_free_page(), the process may risk being locked out in order to
pages of memory to external storage. The memory that has been reserved
not initialized.

After size has been rounded up, an address is found at which the
to be allocated can be mapped to the kernel segment in full. As we have
tioned, in the kernel segment the entire physical memory is mapped from
start, so that the virtual addresses are the same as the physical addresses apa
from an offset dependent on the architecture.

With vmalloc(), the memory to be allocated must be mapped above
end of physical memory, as __get_free_page() (*see* Section 4.4.2) only alloca
individual pages, and not necessarily consecutive ones. In x86 architecture,
search begins at the next address after physical memory, located on an 8 Mby
boundary. The addresses here may already have been allocated by previ
vmalloc calls. One page of memory is left free after each of the reserved area
to cushion accesses exceeding the allocated memory area.

The free pages are mapped by vmalloc() to the address range
memory which has just been located. If it is necessary to generate new pa
tables, these are entered in the *memory map* as reserved pages.

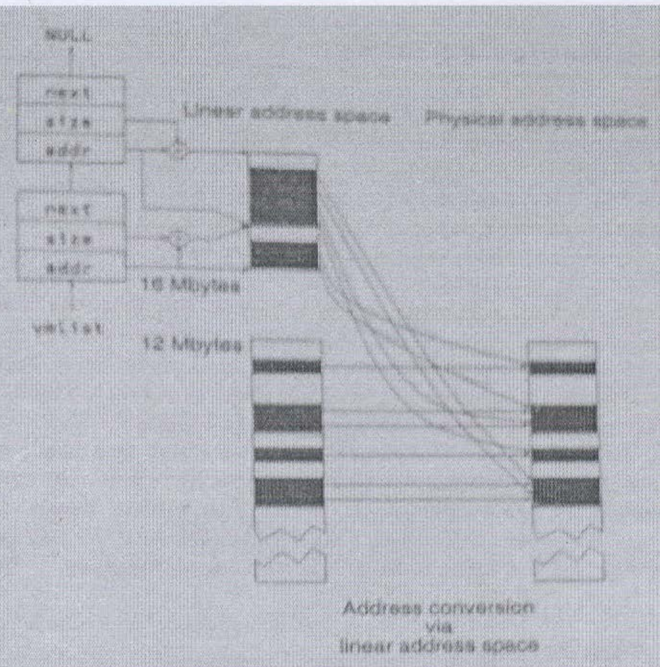


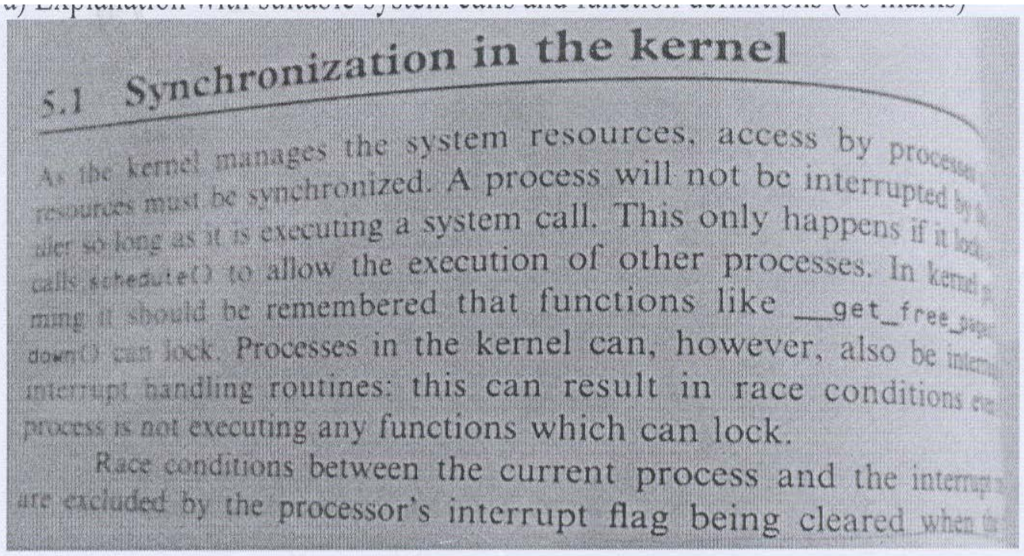
Figure 4.4 Operation of `vmalloc()`.

Kernel addresses set up in this way are managed by LINUX very simply, using a linear list. The related data structure `vm_struct` contains the virtual address of the area and its size, which also includes the page not entered in the page table. As mentioned above, this is intended to intercept cases where the address range is exceeded. This means that the memory area that has been reserved is smaller by one page than the value held in `vm_struct`. As well as this, there is a pointer to the last element in the list and a component `flags` which is not used.

The clear advantage of the `vmalloc()` function is that the size of the area of memory requested can be better adjusted to actual needs than when using `kmalloc()`, which requires 128 Kbytes of consecutive physical memory to reserve just 64 Kbytes. Besides this, `vmalloc()` is limited only by the size of free physical memory and not by its segmentation, as `kmalloc()` is. Since `vmalloc()` does not return any physical addresses and the reserved areas of memory can be spread over non-consecutive pages, this function is not suitable for reserving memory for DMA.

Q.5 a. Show the implementation of synchronization in the LINUX kernel.

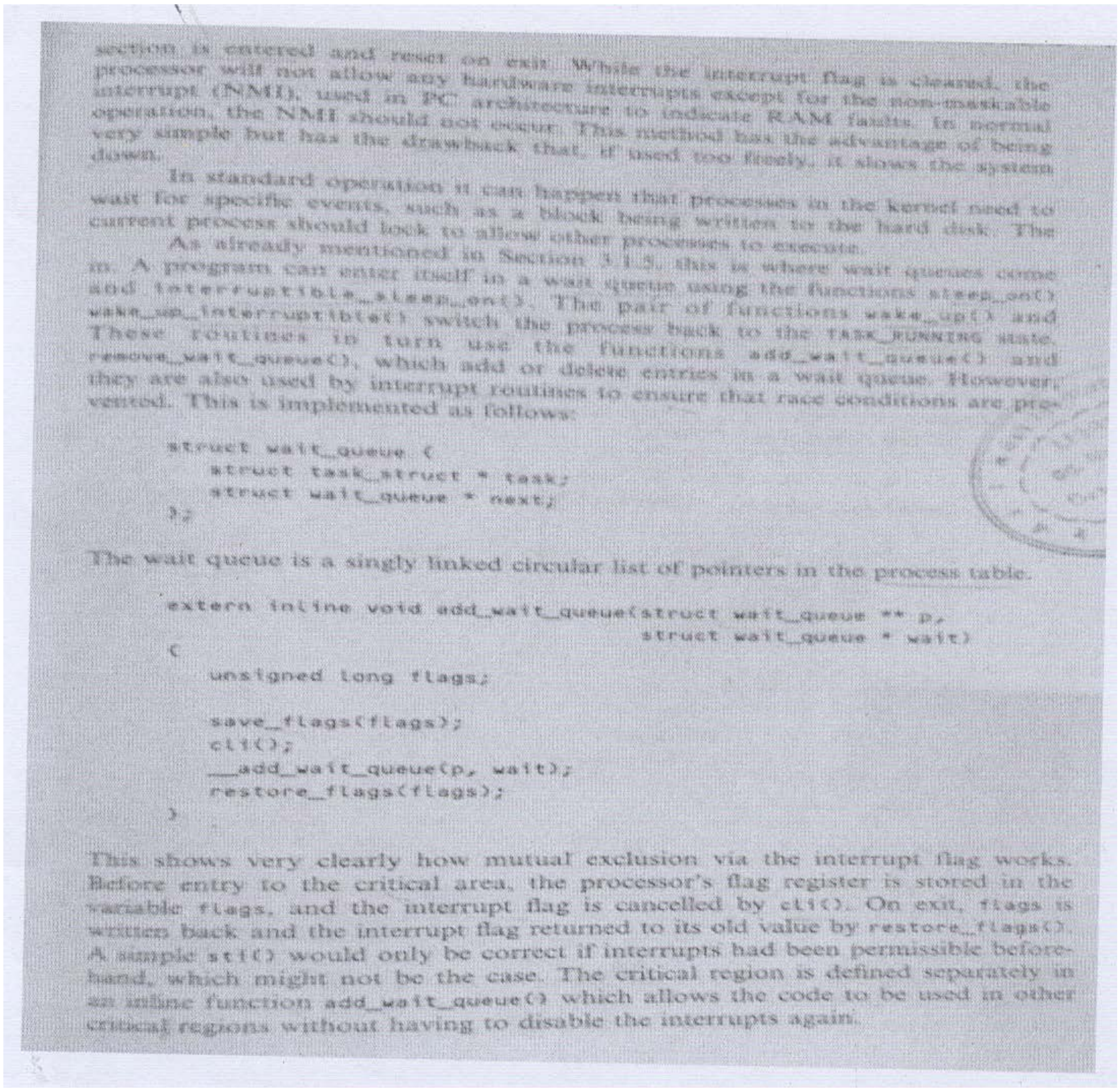
Q5. a) Explanation with suitable system calls and function definitions (10 marks)



5.1 Synchronization in the kernel

As the kernel manages the system resources, access by proces resources must be synchronized. A process will not be interrupted by the uler so long as it is executing a system call. This only happens if it loc calls schedule() to allow the execution of other processes. In kernel p ming it should be remembered that functions like __get_free_pag down() can lock. Processes in the kernel can, however, also be inter interrupt handling routines: this can result in race conditions eve process is not executing any functions which can lock.

Race conditions between the current process and the interrup are excluded by the processor's interrupt flag being cleared when th



section is entered and reset on exit. While the interrupt flag is cleared, the processor will not allow any hardware interrupts except for the non-maskable interrupt (NMI), used in PC architecture to indicate RAM faults. In normal operation, the NMI should not occur. This method has the advantage of being very simple but has the drawback that, if used too freely, it slows the system down.

In standard operation it can happen that processes in the kernel need to wait for specific events, such as a block being written to the hard disk. The current process should lock to allow other processes to execute.

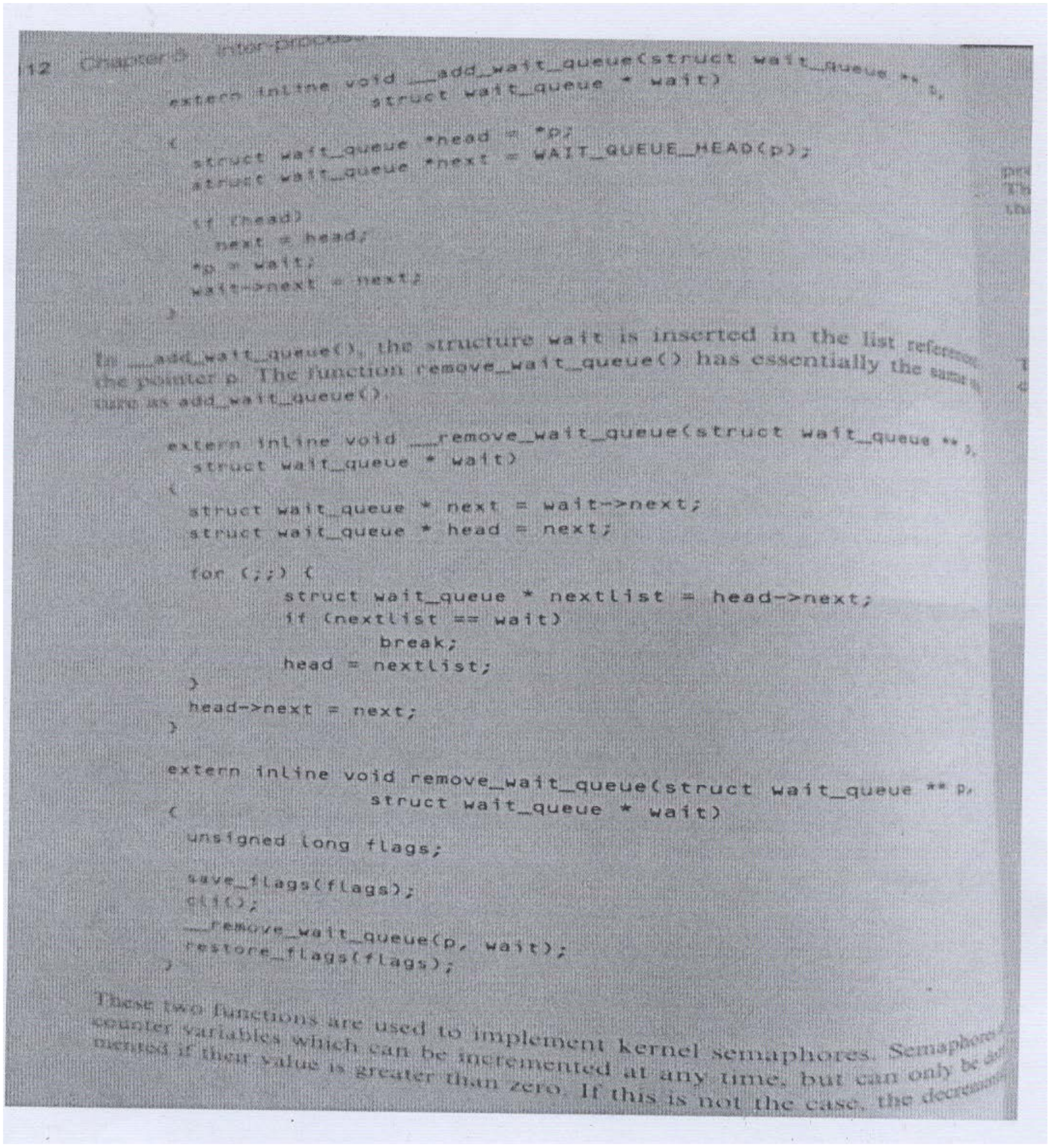
As already mentioned in Section 3.1.5, this is where wait queues come in. A program can enter itself in a wait queue using the functions sleep_on() and interruptible_sleep_on(). The pair of functions wake_up() and wake_up_interruptible() switch the process back to the TASK_RUNNING state. These routines in turn use the functions add_wait_queue() and remove_wait_queue(), which add or delete entries in a wait queue. However, they are also used by interrupt routines to ensure that race conditions are prevented. This is implemented as follows:

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

The wait queue is a singly linked circular list of pointers in the process table.

```
extern inline void add_wait_queue(struct wait_queue ** p,
                                  struct wait_queue * wait)
{
    unsigned long flags;

    save_flags(flags);
    cli();
    __add_wait_queue(p, wait);
    restore_flags(flags);
}
```

This shows very clearly how mutual exclusion via the interrupt flag works. Before entry to the critical area, the processor's flag register is stored in the variable flags, and the interrupt flag is cancelled by cli(). On exit, flags is written back and the interrupt flag returned to its old value by restore_flags(). A simple sti() would only be correct if interrupts had been permissible beforehand, which might not be the case. The critical region is defined separately in an inline function add_wait_queue() which allows the code to be used in other critical regions without having to disable the interrupts again.

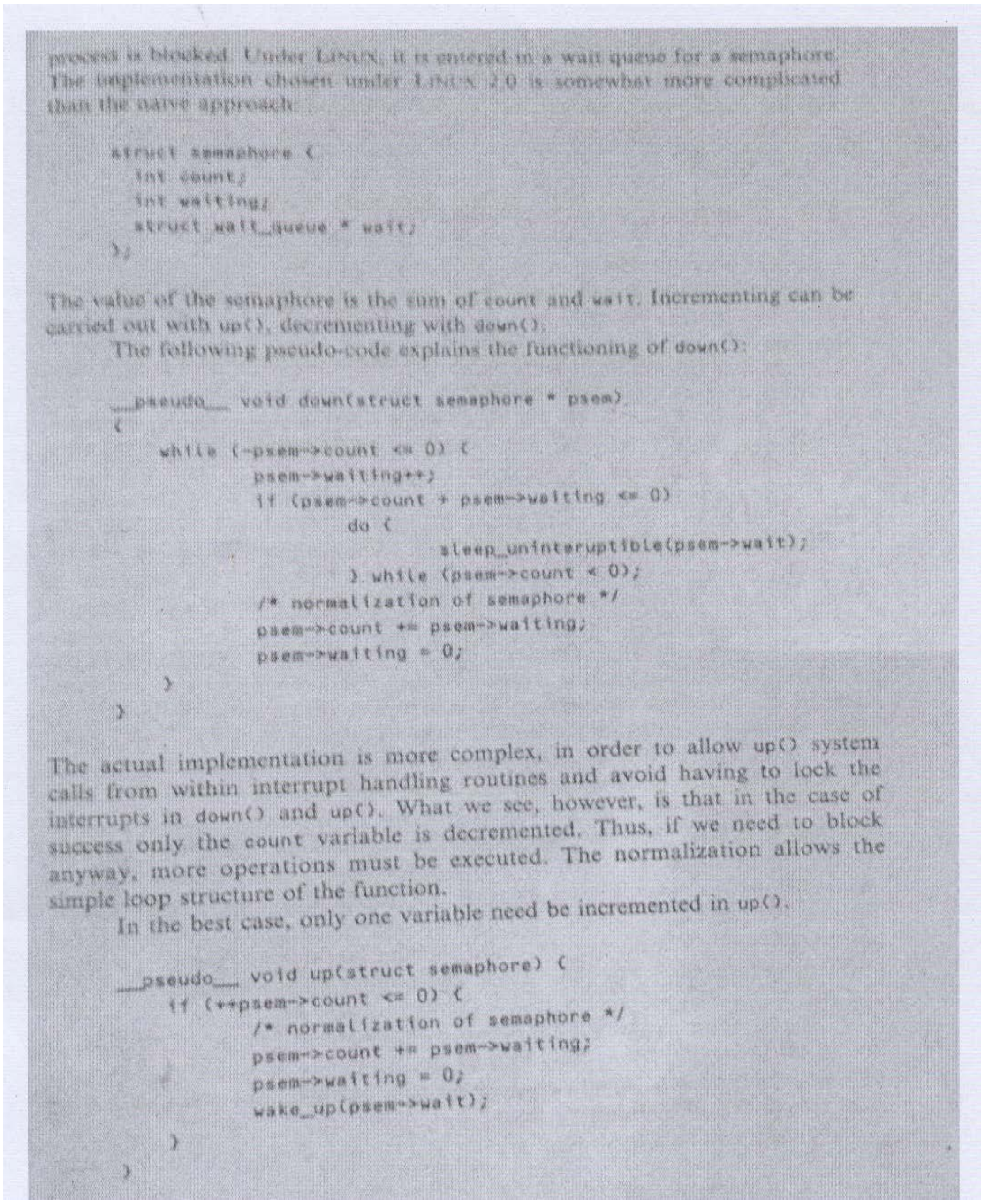


```
extern inline void __add_wait_queue(struct wait_queue ** p,
                struct wait_queue * wait)
{
  struct wait_queue *head = *p;
  struct wait_queue *next = WAIT_QUEUE_HEAD(p);

  if (head)
    next = head;
  *p = wait;
  wait->next = next;
}
```

In __add_wait_queue(), the structure wait is inserted in the list referen
the pointer p. The function remove_wait_queue() has essentially the same
ture as add_wait_queue().

```
extern inline void __remove_wait_queue(struct wait_queue ** p,
  struct wait_queue * wait)
{
  struct wait_queue * next = wait->next;
  struct wait_queue * head = next;

  for (;;) {
          struct wait_queue * nextlist = head->next;
          if (nextlist == wait)
                  break;
          head = nextlist;
  }
  head->next = next;
}

extern inline void remove_wait_queue(struct wait_queue ** p,
                struct wait_queue * wait)
{
  unsigned long flags;

  save_flags(flags);
  cli();
  __remove_wait_queue(p, wait);
  restore_flags(flags);
}
```

These two functions are used to implement kernel semaphores. Semaphore
counter variables which can be incremented at any time, but can only be de
mented if their value is greater than zero. If this is not the case, the decrem



process is blocked. Under LINUX, it is entered in a wait queue for a semaphore. The implementation chosen under LINUX 2.0 is somewhat more complicated than the naive approach.

```
struct semaphore {
  int count;
  int waiting;
  struct wait_queue * wait;
};
```

The value of the semaphore is the sum of count and wait. Incrementing can be carried out with up(), decrementing with down().

The following pseudo-code explains the functioning of down():

```
__pseudo__ void down(struct semaphore * psem)
{
    while (-psem->count <= 0) {
            psem->waiting++;
            if (psem->count + psem->waiting <= 0)
                    do {
                            sleep_uninteruptible(psem->wait);
                    } while (psem->count < 0);
            /* normalization of semaphore */
            psem->count += psem->waiting;
            psem->waiting = 0;
    }
}
```

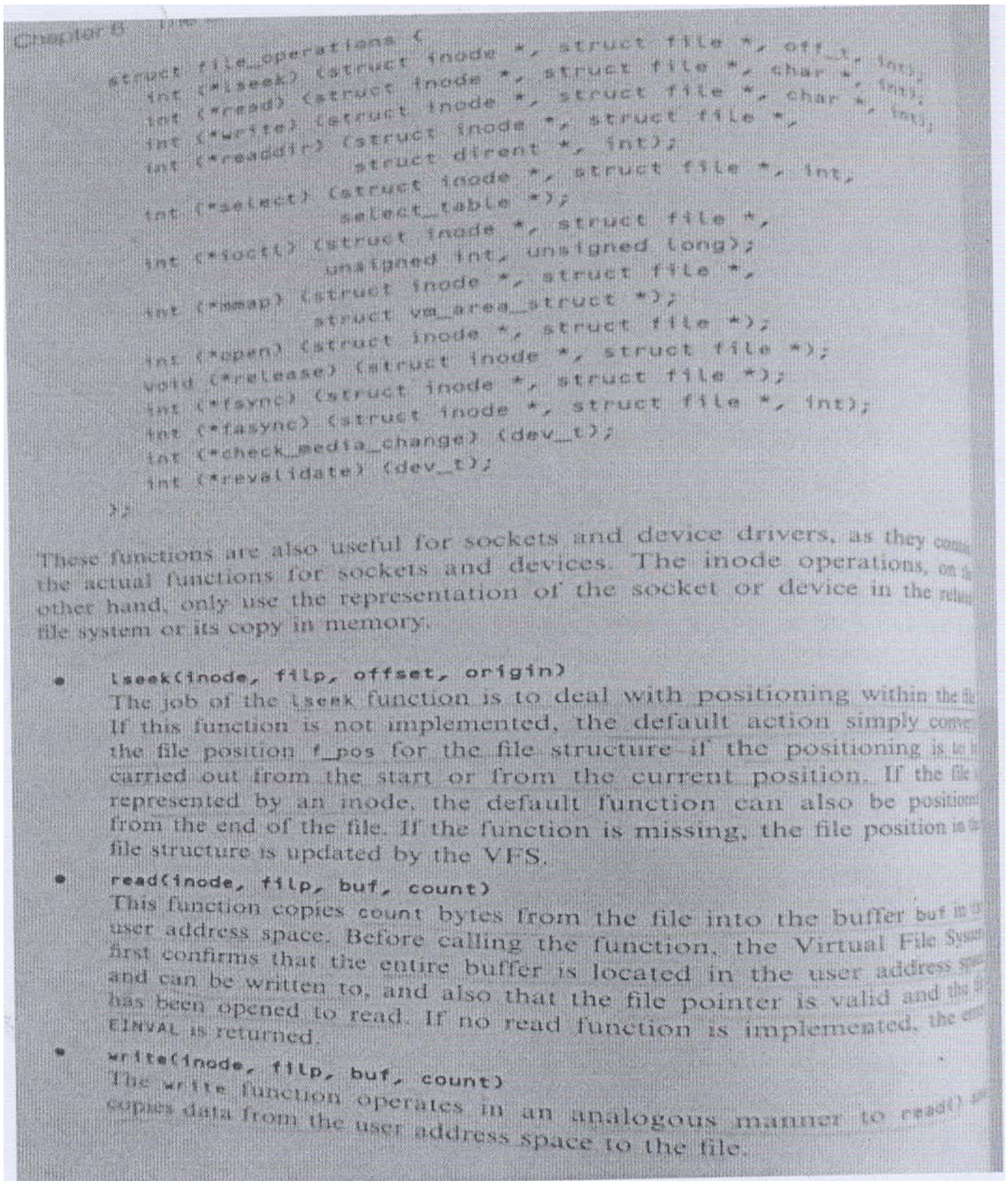
The actual implementation is more complex, in order to allow up() system calls from within interrupt handling routines and avoid having to lock the interrupts in down() and up(). What we see, however, is that in the case of success only the count variable is decremented. Thus, if we need to block anyway, more operations must be executed. The normalization allows the simple loop structure of the function.

In the best case, only one variable need be incremented in up().

```
__pseudo__ void up(struct semaphore) {
    if (++psem->count <= 0) {
            /* normalization of semaphore */
            psem->count += psem->waiting;
            psem->waiting = 0;
            wake_up(psem->wait);
    }
}
```

b.How is inter-process communication achieved in Linux? Explain.

Q.6a. Write a note on any four file operations in LINUX.

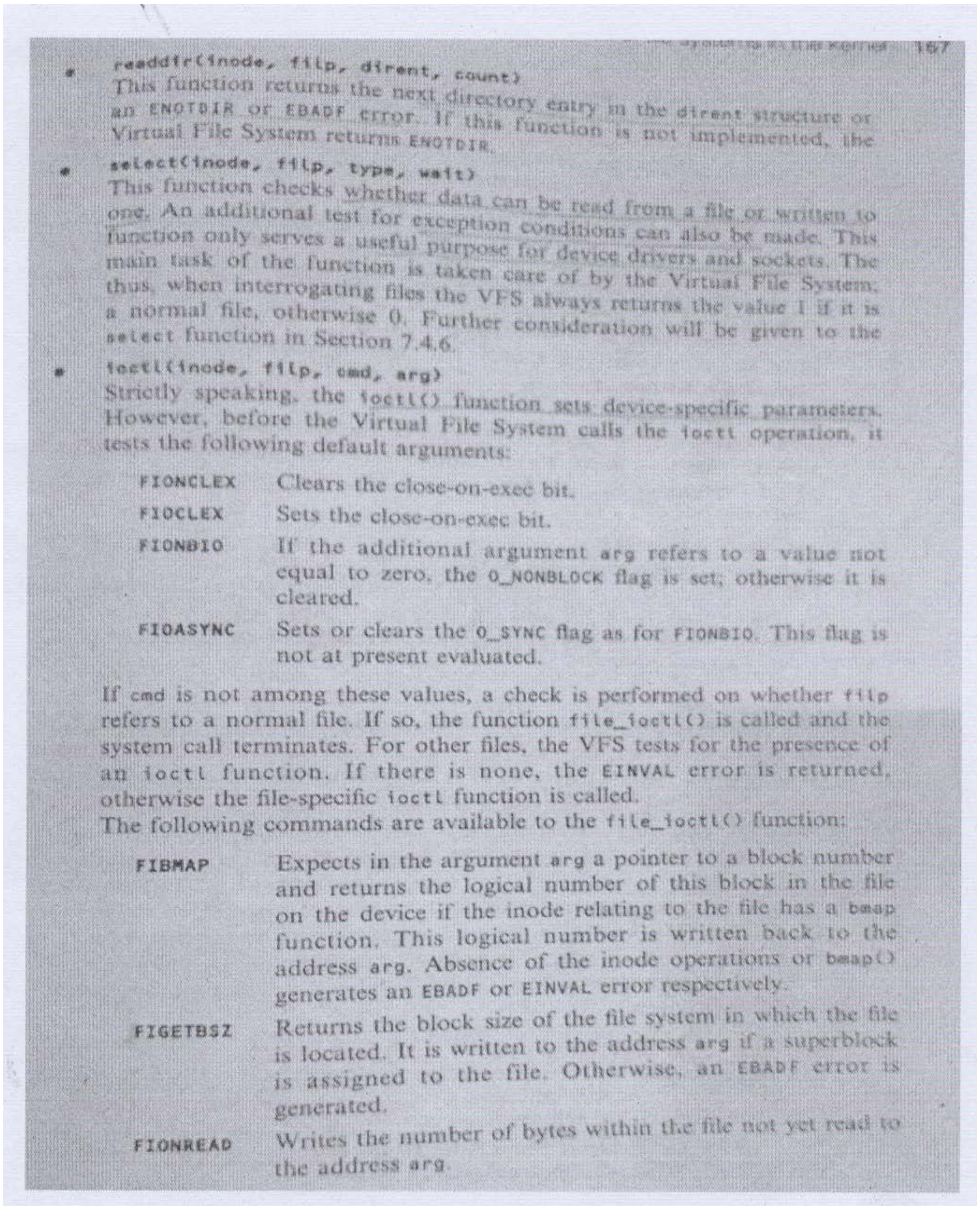


Chapter 8

```
struct file_operations {
    int (*lseek) (struct inode *, struct file *, off_t, int);
    int (*read) (struct inode *, struct file *, char *, int);
    int (*write) (struct inode *, struct file *, char *, int);
    int (*readdir) (struct inode *, struct file *,
                    struct dirent *, int);
    int (*select) (struct inode *, struct file *, int,
                   select_table *);
    int (*ioctl) (struct inode *, struct file *,
                  unsigned int, unsigned long);
    int (*mmap) (struct inode *, struct file *,
                 struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    void (*release) (struct inode *, struct file *);
    int (*fsync) (struct inode *, struct file *);
    int (*fasync) (struct inode *, struct file *, int);
    int (*check_media_change) (dev_t);
    int (*revalidate) (dev_t);
};
```

These functions are also useful for sockets and device drivers, as they con the actual functions for sockets and devices. The inode operations, on th other hand, only use the representation of the socket or device in the rele file system or its copy in memory.

- `lseek(inode, filp, offset, origin)`
  The job of the `lseek` function is to deal with positioning within the fi If this function is not implemented, the default action simply conve the file position `f_pos` for the file structure if the positioning is to b carried out from the start or from the current position. If the file represented by an inode, the default function can also be position from the end of the file. If the function is missing, the file position in th file structure is updated by the VFS.
- `read(inode, filp, buf, count)`
  This function copies count bytes from the file into the buffer `buf` in th user address space. Before calling the function, the Virtual File Syst first confirms that the entire buffer is located in the user address sp and can be written to, and also that the file pointer is valid and the fi has been opened to read. If no read function is implemented, the err `EINVAL` is returned.
- `write(inode, filp, buf, count)`
  The `write` function operates in an analogous manner to `read()` an copies data from the user address space to the file.

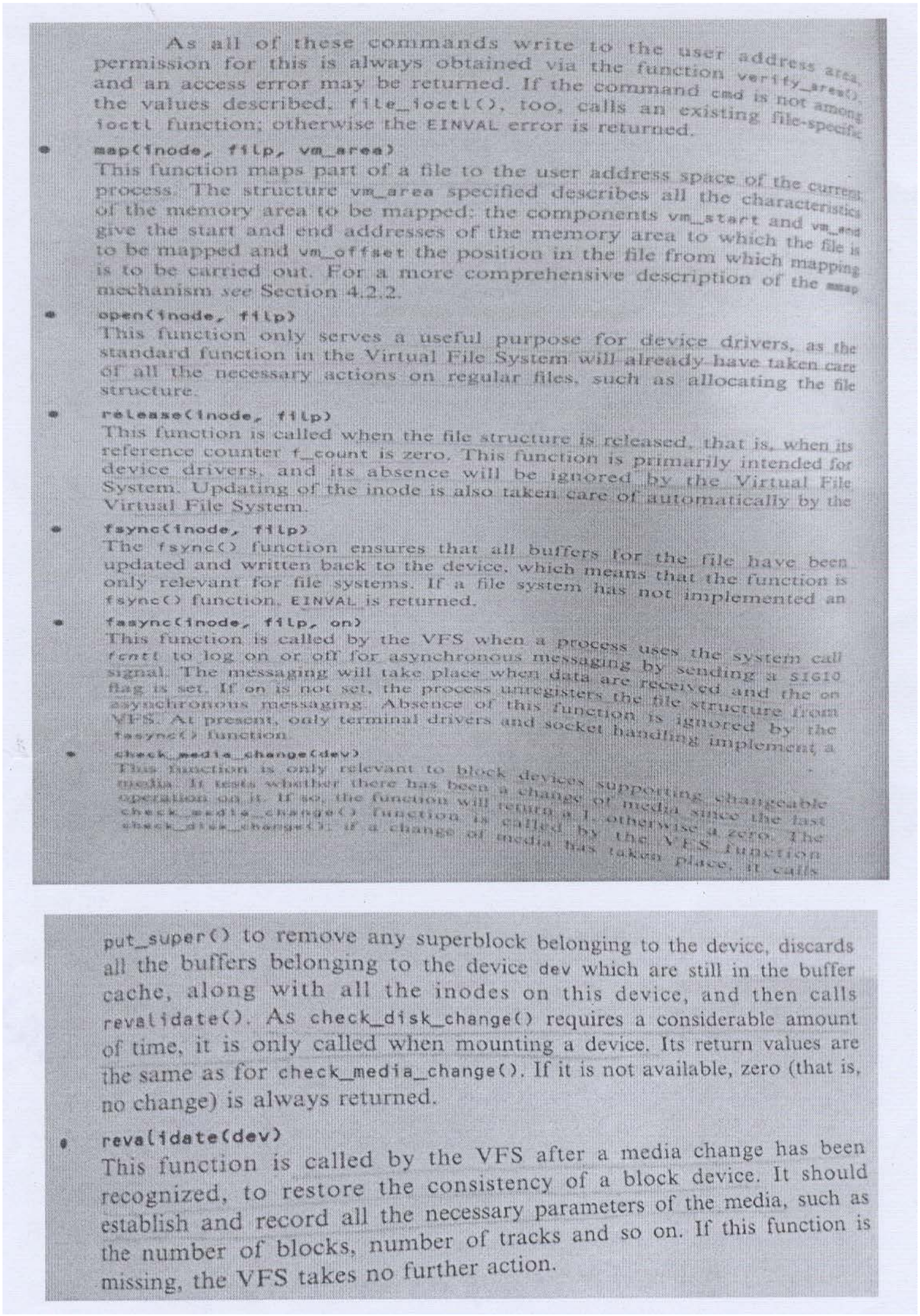


- `readdir(inode, filp, dirent, count)`
  This function returns the next directory entry in the `dirent` structure or an `ENOTDIR` or `EBADF` error. If this function is not implemented, the Virtual File System returns `ENOTDIR`.
- `select(inode, filp, type, wait)`
  This function checks whether data can be read from a file or written to one. An additional test for exception conditions can also be made. This function only serves a useful purpose for device drivers and sockets. The main task of the function is taken care of by the Virtual File System; thus, when interrogating files the VFS always returns the value 1 if it is a normal file, otherwise 0. Further consideration will be given to the `select` function in Section 7.4.6.
- `ioctl(inode, filp, cmd, arg)`
  Strictly speaking, the `ioctl()` function sets device-specific parameters. However, before the Virtual File System calls the `ioctl` operation, it tests the following default arguments:

`FIONCLEX`	Clears the close-on-exec bit.
`FIOCLEX`	Sets the close-on-exec bit.
`FIONBIO`	If the additional argument `arg` refers to a value not equal to zero, the `O_NONBLOCK` flag is set; otherwise it is cleared.
`FIOASYNC`	Sets or clears the `O_SYNC` flag as for `FIONBIO`. This flag is not at present evaluated.

If `cmd` is not among these values, a check is performed on whether `filp` refers to a normal file. If so, the function `file_ioctl()` is called and the system call terminates. For other files, the VFS tests for the presence of an `ioctl` function. If there is none, the `EINVAL` error is returned, otherwise the file-specific `ioctl` function is called.
The following commands are available to the `file_ioctl()` function:

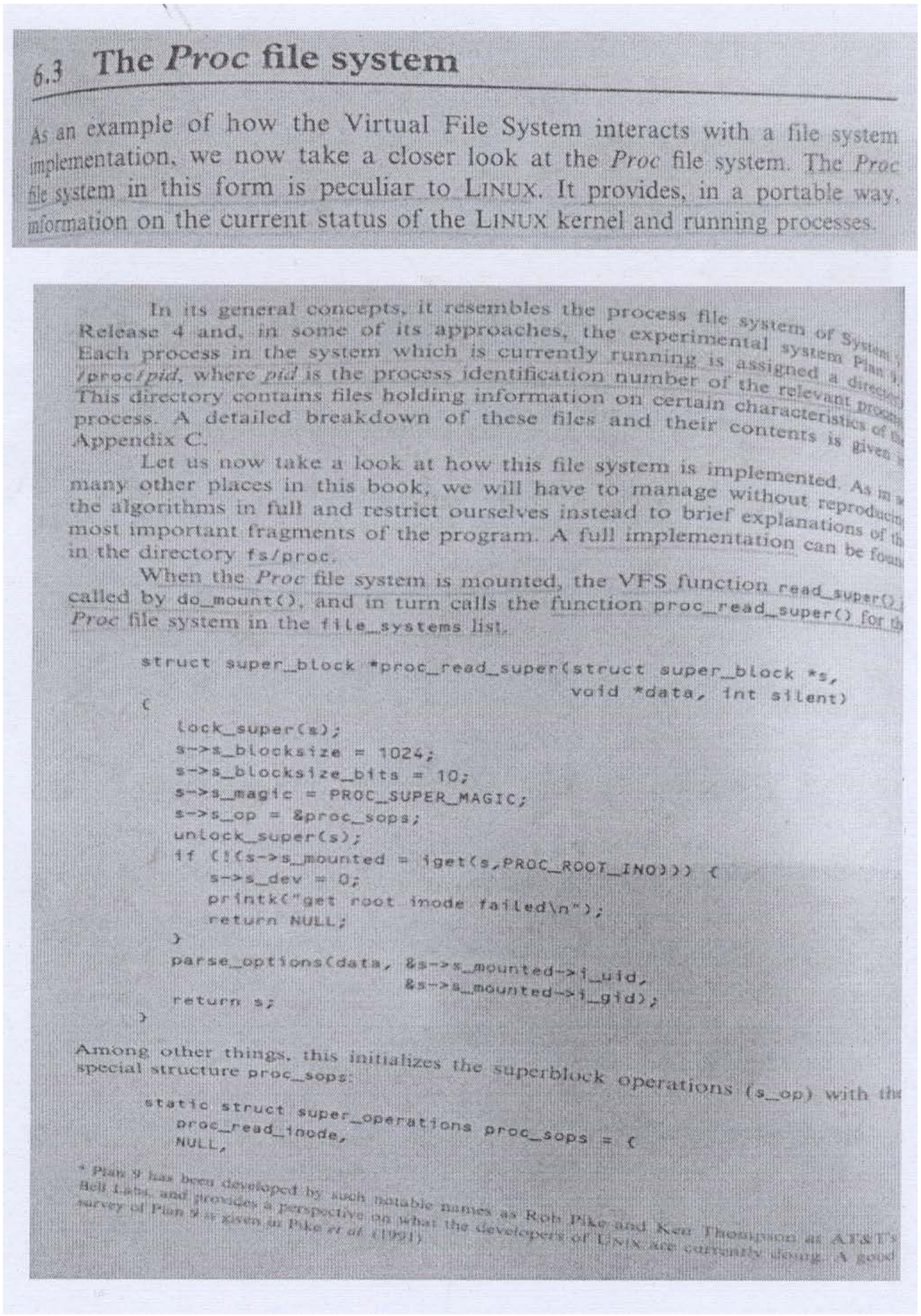
`FIBMAP`	Expects in the argument `arg` a pointer to a block number and returns the logical number of this block in the file on the device if the inode relating to the file has a `bmap` function. This logical number is written back to the address `arg`. Absence of the inode operations or `bmap()` generates an `EBADF` or `EINVAL` error respectively.
`FIGETBSZ`	Returns the block size of the file system in which the file is located. It is written to the address `arg` if a superblock is assigned to the file. Otherwise, an `EBADF` error is generated.
`FIONREAD`	Writes the number of bytes within the file not yet read to the address `arg`.



As all of these commands write to the user address area, permission for this is always obtained via the function verify_area(), and an access error may be returned. If the command cmd is not among the values described, file_ioctl(), too, calls an existing file-specific ioctl function; otherwise the EINVAL error is returned.

- map(inode, filp, vm_area)
  This function maps part of a file to the user address space of the current process. The structure vm_area specified describes all the characteristics of the memory area to be mapped: the components vm_start and vm_end give the start and end addresses of the memory area to which the file is to be mapped and vm_offset the position in the file from which mapping is to be carried out. For a more comprehensive description of the mmap mechanism *see* Section 4.2.2.
- open(inode, filp)
  This function only serves a useful purpose for device drivers, as the standard function in the Virtual File System will already have taken care of all the necessary actions on regular files, such as allocating the file structure.
- release(inode, filp)
  This function is called when the file structure is released, that is, when its reference counter f_count is zero. This function is primarily intended for device drivers, and its absence will be ignored by the Virtual File System. Updating of the inode is also taken care of automatically by the Virtual File System.
- fsync(inode, filp)
  The fsync() function ensures that all buffers for the file have been updated and written back to the device, which means that the function is only relevant for file systems. If a file system has not implemented an fsync() function, EINVAL is returned.
- fasync(inode, filp, on)
  This function is called by the VFS when a process uses the system call *fcntl* to log on or off for asynchronous messaging by sending a SIGIO signal. The messaging will take place when data are received and the on flag is set. If on is not set, the process unregisters the file structure from asynchronous messaging. Absence of this function is ignored by the VFS. At present, only terminal drivers and socket handling implement a fasync() function.
- check_media_change(dev)
  This function is only relevant to block devices supporting changeable media. It tests whether there has been a change of media since the last operation on it. If so, the function will return a 1, otherwise a zero. The check_media_change() function is called by the VFS function check_disk_change(); if a change of media has taken place, it calls put_super() to remove any superblock belonging to the device, discards all the buffers belonging to the device dev which are still in the buffer cache, along with all the inodes on this device, and then calls revalidate(). As check_disk_change() requires a considerable amount of time, it is only called when mounting a device. Its return values are the same as for check_media_change(). If it is not available, zero (that is, no change) is always returned.
- revalidate(dev)
  This function is called by the VFS after a media change has been recognized, to restore the consistency of a block device. It should establish and record all the necessary parameters of the media, such as the number of blocks, number of tracks and so on. If this function is missing, the VFS takes no further action.

b.Explain, how the Virtual File System interacts with a file system implementation?



6.3 The *Proc* file system

As an example of how the Virtual File System interacts with a file system implementation, we now take a closer look at the *Proc* file system. The *Proc* file system in this form is peculiar to LINUX. It provides, in a portable way, information on the current status of the LINUX kernel and running processes.

In its general concepts, it resembles the process file system of System Release 4 and, in some of its approaches, the experimental system Plan 9. Each process in the system which is currently running is assigned a directory */proc/pid*, where *pid* is the process identification number of the relevant process. This directory contains files holding information on certain characteristics of the process. A detailed breakdown of these files and their contents is given in Appendix C.

Let us now take a look at how this file system is implemented. As in many other places in this book, we will have to manage without reproducing the algorithms in full and restrict ourselves instead to brief explanations of the most important fragments of the program. A full implementation can be found in the directory `fs/proc`.

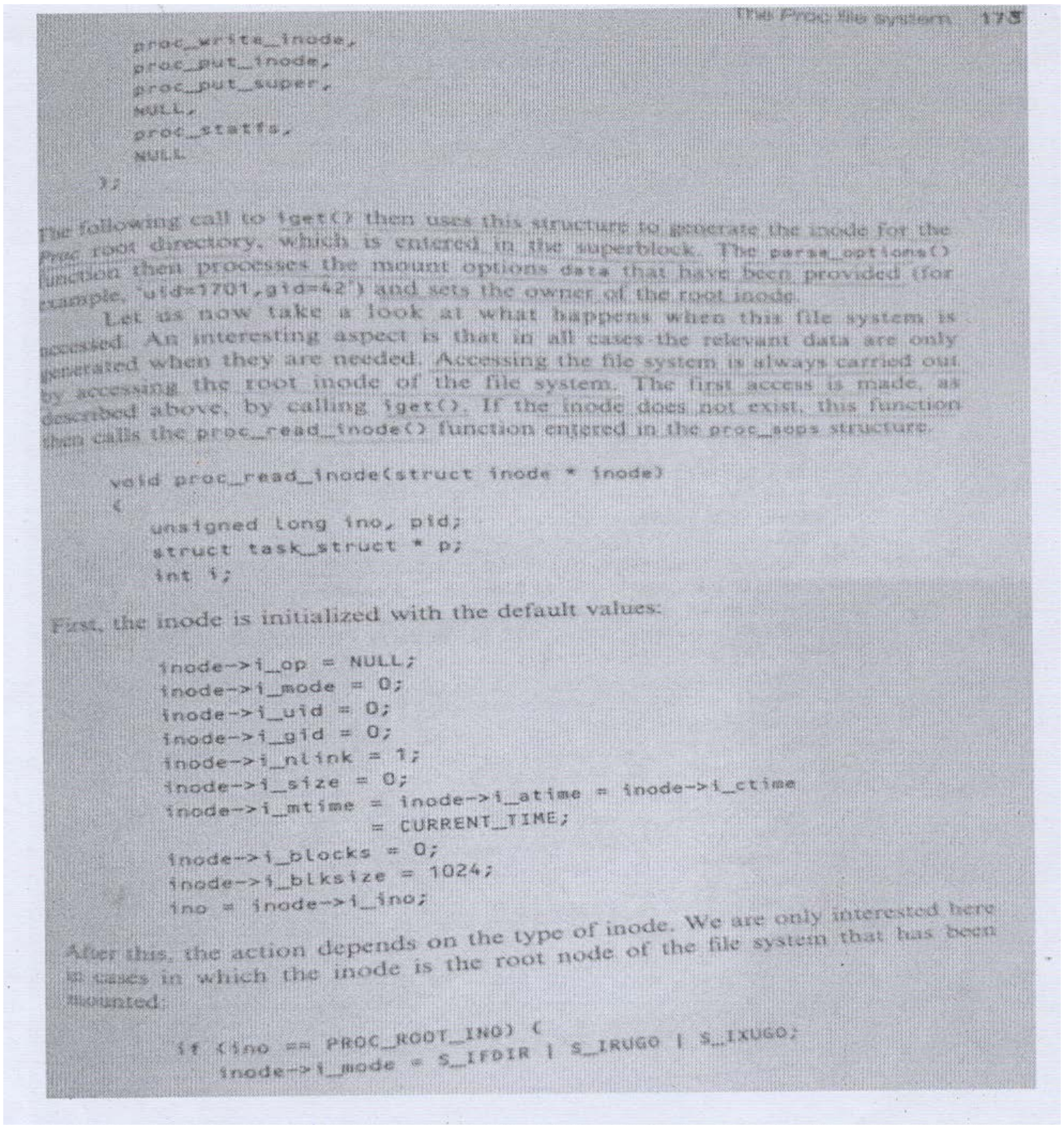
When the *Proc* file system is mounted, the VFS function `read_super()` is called by `do_mount()`, and in turn calls the function `proc_read_super()` for the *Proc* file system in the `file_systems` list.

```
struct super_block *proc_read_super(struct super_block *s,
                                    void *data, int silent)
{
   lock_super(s);
   s->s_blocksize = 1024;
   s->s_blocksize_bits = 10;
   s->s_magic = PROC_SUPER_MAGIC;
   s->s_op = &proc_sops;
   unlock_super(s);
   if (!(s->s_mounted = iget(s,PROC_ROOT_INO))) {
      s->s_dev = 0;
      printk("get root inode failed\n");
      return NULL;
   }
   parse_options(data, &s->s_mounted->i_uid,
                       &s->s_mounted->i_gid);
   return s;
}
```

Among other things, this initializes the superblock operations (`s_op`) with the special structure `proc_sops`:

```
static struct super_operations proc_sops = {
   proc_read_inode,
   NULL,
```

* Plan 9 has been developed by such notable names as Rob Pike and Ken Thompson at AT&T's Bell Labs, and provides a perspective on what the developers of UNIX are currently doing. A good survey of Plan 9 is given in Pike *et al.* (1991).



```
        proc_write_inode,
        proc_put_inode,
        proc_put_super,
        NULL,
        proc_statfs,
        NULL
    };
```

The following call to `iget()` then uses this structure to generate the inode for the Proc root directory, which is entered in the superblock. The `parse_options()` function then processes the mount options `data` that have been provided (for example, '`uid=1701,gid=42`') and sets the owner of the root inode.

Let us now take a look at what happens when this file system is accessed. An interesting aspect is that in all cases the relevant data are only generated when they are needed. Accessing the file system is always carried out by accessing the root inode of the file system. The first access is made, as described above, by calling `iget()`. If the inode does not exist, this function then calls the `proc_read_inode()` function entered in the `proc_sops` structure.

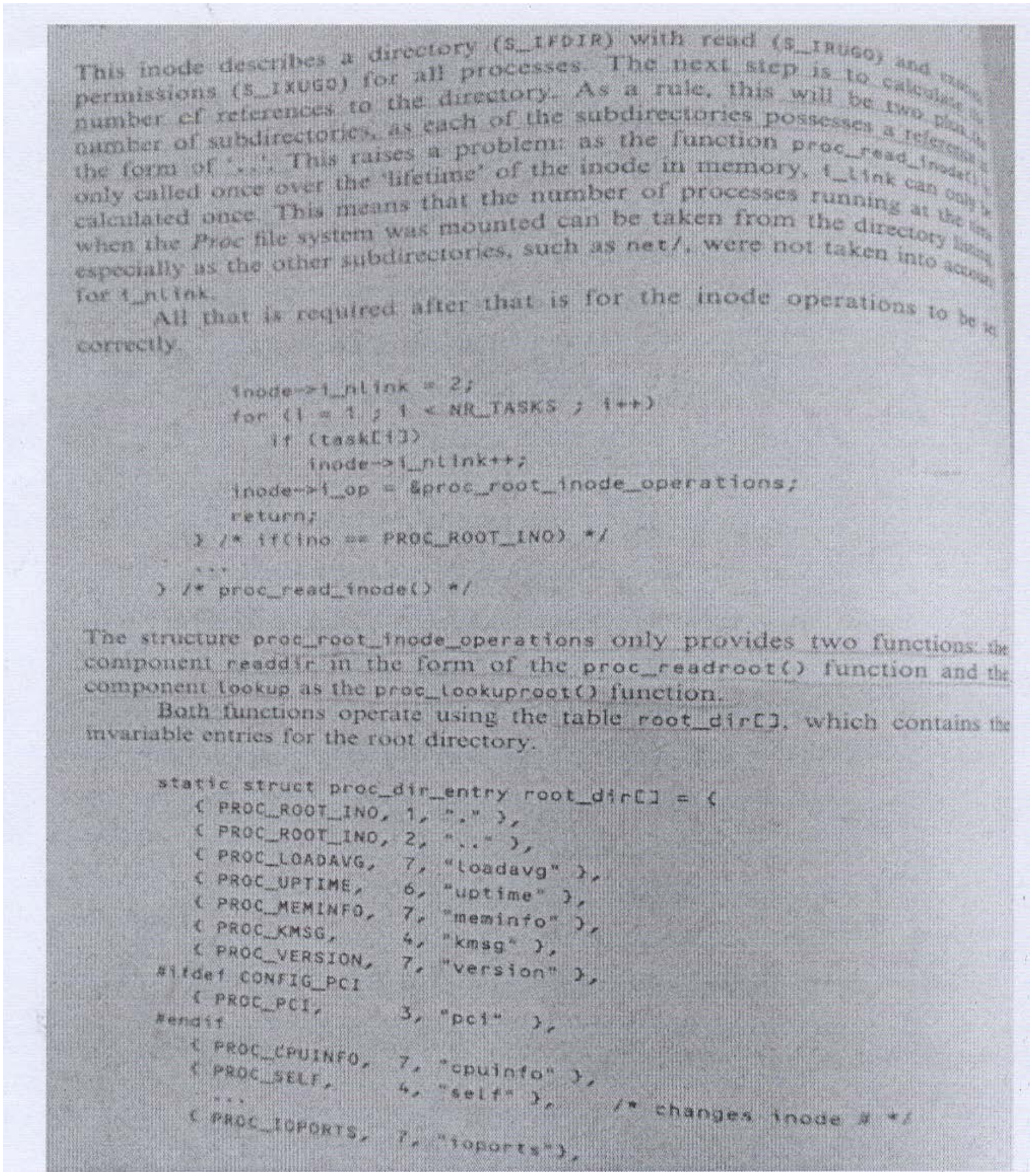
```
    void proc_read_inode(struct inode * inode)
    {
        unsigned long ino, pid;
        struct task_struct * p;
        int i;
```

First, the inode is initialized with the default values:

```
        inode->i_op = NULL;
        inode->i_mode = 0;
        inode->i_uid = 0;
        inode->i_gid = 0;
        inode->i_nlink = 1;
        inode->i_size = 0;
        inode->i_mtime = inode->i_atime = inode->i_ctime
                       = CURRENT_TIME;
        inode->i_blocks = 0;
        inode->i_blksize = 1024;
        ino = inode->i_ino;
```

After this, the action depends on the type of inode. We are only interested here in cases in which the inode is the root node of the file system that has been mounted:

```
        if (ino == PROC_ROOT_INO) {
            inode->i_mode = S_IFDIR | S_IRUGO | S_IXUGO;
```



This inode describes a directory (S_IFDIR) with read (S_IRUGO) and execute permissions (S_IXUGO) for all processes. The next step is to calculate the number of references to the directory. As a rule, this will be two plus the number of subdirectories, as each of the subdirectories possesses a reference in the form of '..'. This raises a problem: as the function proc_read_inode() is only called once over the 'lifetime' of the inode in memory, i_link can only be calculated once. This means that the number of processes running at the time when the *Proc* file system was mounted can be taken from the directory listing, especially as the other subdirectories, such as net/, were not taken into account for i_nlink.

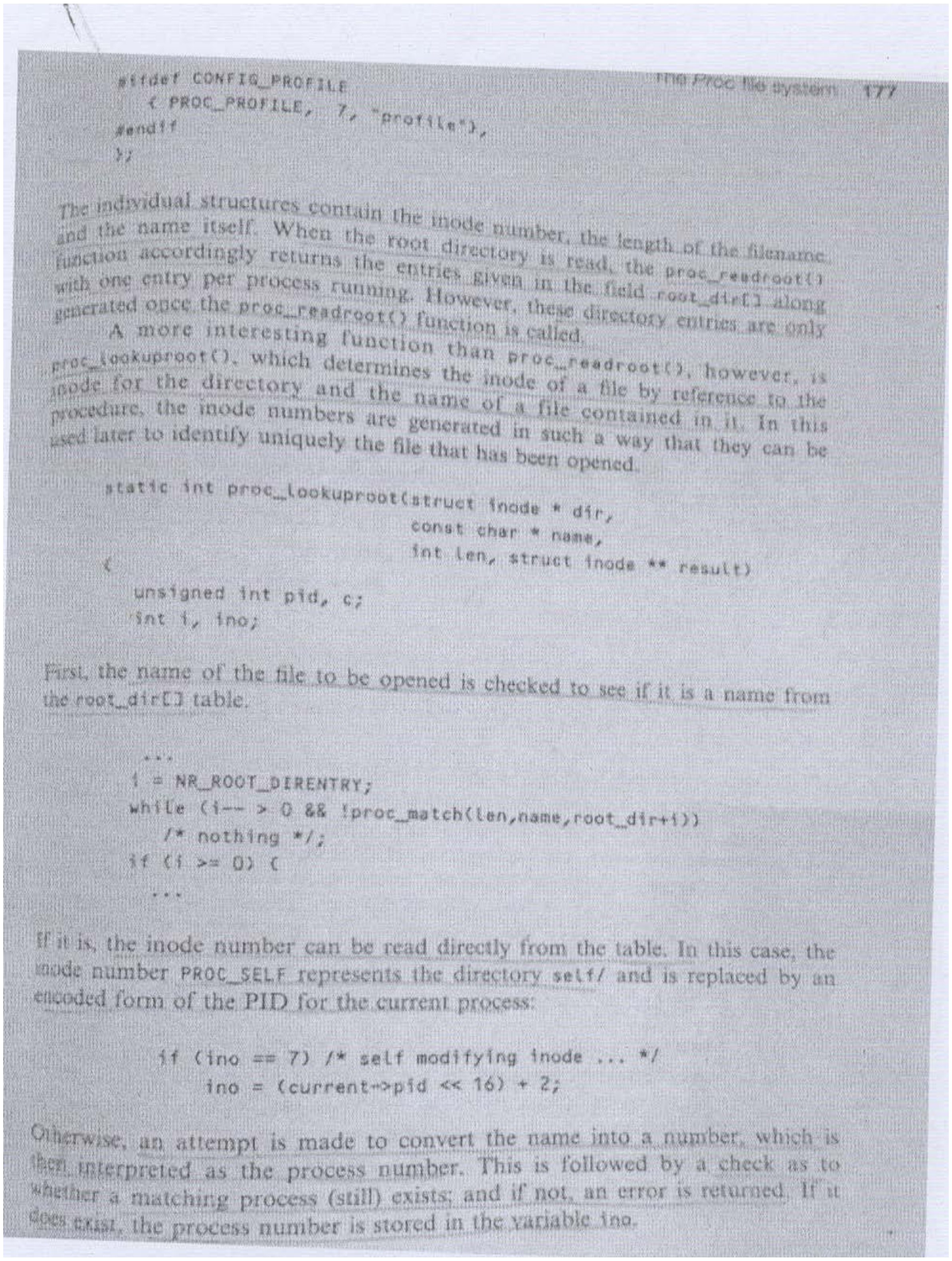
All that is required after that is for the inode operations to be set correctly.

```
            inode->i_nlink = 2;
            for (i = 1 ; i < NR_TASKS ; i++)
                if (task[i])
                    inode->i_nlink++;
            inode->i_op = &proc_root_inode_operations;
            return;
        } /* if(ino == PROC_ROOT_INO) */
        ...
    } /* proc_read_inode() */
```

The structure proc_root_inode_operations only provides two functions: the component readdir in the form of the proc_readroot() function and the component lookup as the proc_lookuproot() function.

Both functions operate using the table root_dir[], which contains the invariable entries for the root directory.

```
static struct proc_dir_entry root_dir[] = {
    { PROC_ROOT_INO, 1, "." },
    { PROC_ROOT_INO, 2, ".." },
    { PROC_LOADAVG,  7, "loadavg" },
    { PROC_UPTIME,   6, "uptime" },
    { PROC_MEMINFO,  7, "meminfo" },
    { PROC_KMSG,     4, "kmsg" },
    { PROC_VERSION,  7, "version" },
#ifdef CONFIG_PCI
    { PROC_PCI,      3, "pci"  },
#endif
    { PROC_CPUINFO,  7, "cpuinfo" },
    { PROC_SELF,     4, "self" },      /* changes inode # */
    ...
    { PROC_IOPORTS,  7, "ioports"},
```



```
#ifdef CONFIG_PROFILE
   { PROC_PROFILE,  7, "profile"},
#endif
};
```

The individual structures contain the inode number, the length of the filename and the name itself. When the root directory is read, the `proc_readroot()` function accordingly returns the entries given in the field `root_dir[]` along with one entry per process running. However, these directory entries are only generated once the `proc_readroot()` function is called.

A more interesting function than `proc_readroot()`, however, is `proc_lookuproot()`, which determines the inode of a file by reference to the inode for the directory and the name of a file contained in it. In this procedure, the inode numbers are generated in such a way that they can be used later to identify uniquely the file that has been opened.

```
static int proc_lookuproot(struct inode * dir,
                           const char * name,
                           int len, struct inode ** result)
{
   unsigned int pid, c;
   int i, ino;
```

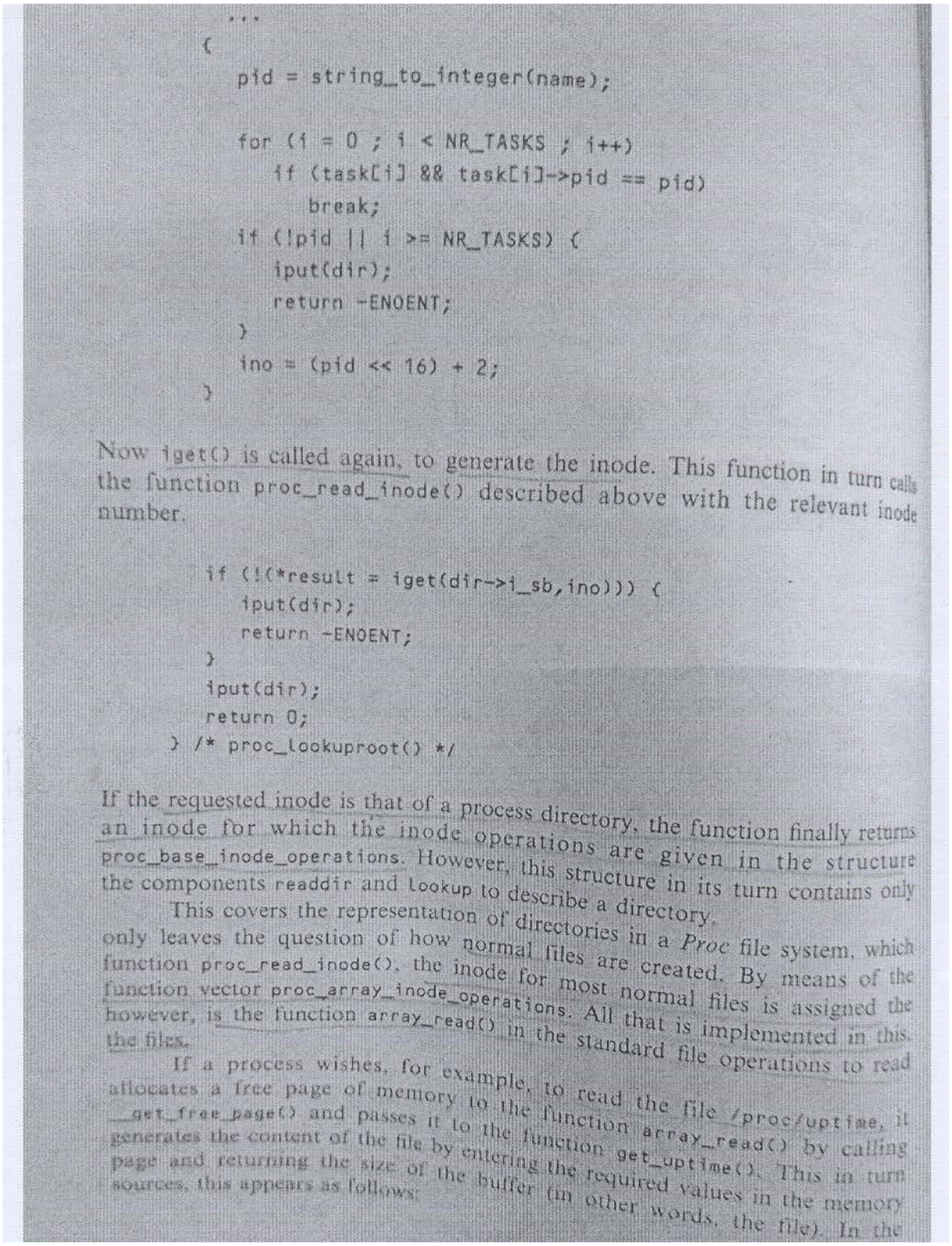
First, the name of the file to be opened is checked to see if it is a name from the `root_dir[]` table.

```
   ...
   i = NR_ROOT_DIRENTRY;
   while (i-- > 0 && !proc_match(len,name,root_dir+i))
      /* nothing */;
   if (i >= 0) {
   ...
```

If it is, the inode number can be read directly from the table. In this case, the inode number `PROC_SELF` represents the directory `self/` and is replaced by an encoded form of the PID for the current process:

```
      if (ino == 7) /* self modifying inode ... */
         ino = (current->pid << 16) + 2;
```

Otherwise, an attempt is made to convert the name into a number, which is then interpreted as the process number. This is followed by a check as to whether a matching process (still) exists; and if not, an error is returned. If it does exist, the process number is stored in the variable `ino`.



```
    ...
    {
        pid = string_to_integer(name);

        for (i = 0 ; i < NR_TASKS ; i++)
            if (task[i] && task[i]->pid == pid)
                break;
        if (!pid || i >= NR_TASKS) {
            iput(dir);
            return -ENOENT;
        }
        ino = (pid << 16) + 2;
    }
```

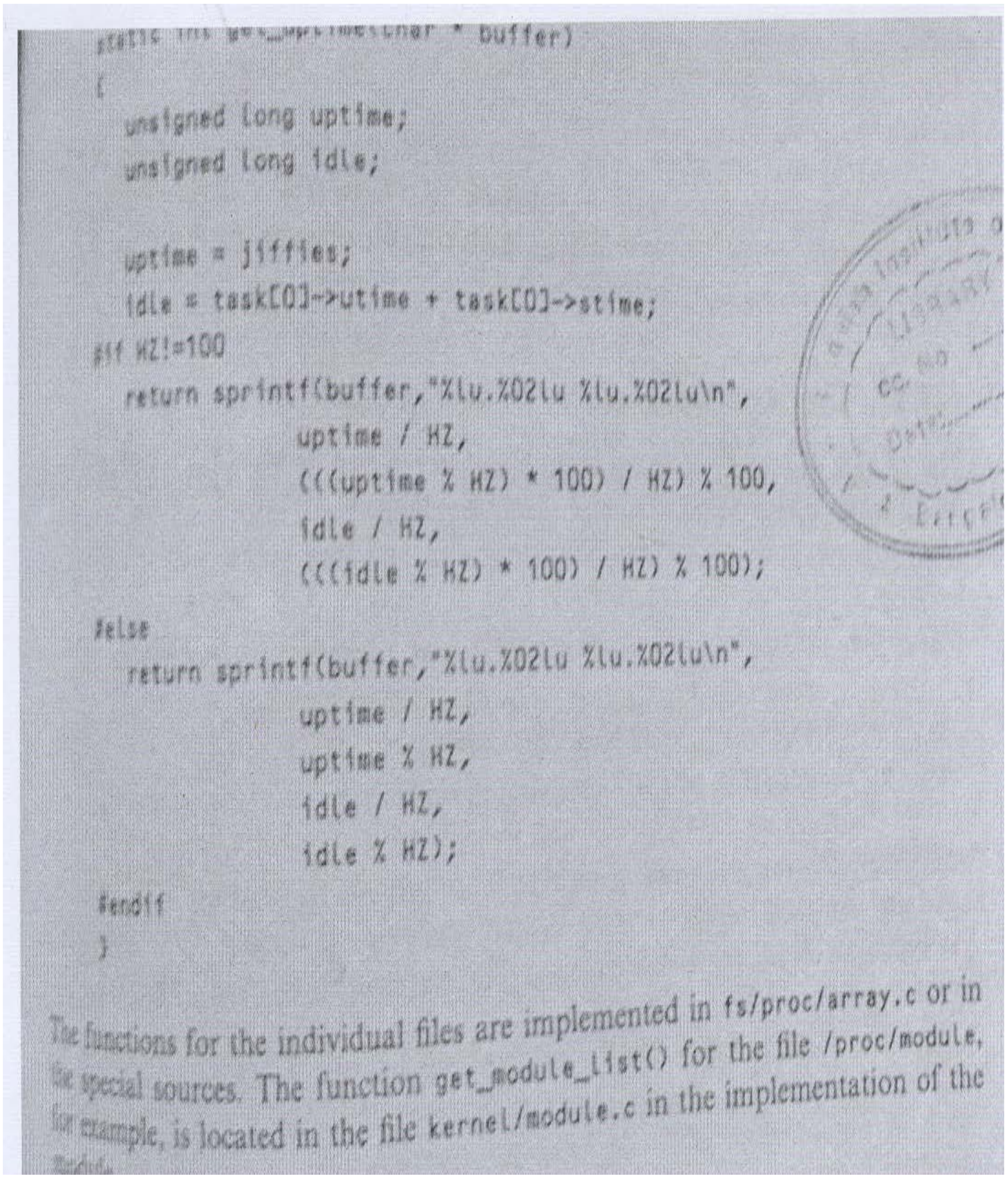
Now iget() is called again, to generate the inode. This function in turn calls the function proc_read_inode() described above with the relevant inode number.

```
    if (!(*result = iget(dir->i_sb,ino))) {
        iput(dir);
        return -ENOENT;
    }
    iput(dir);
    return 0;
} /* proc_lookuproot() */
```

If the requested inode is that of a process directory, the function finally returns an inode for which the inode operations are given in the structure proc_base_inode_operations. However, this structure in its turn contains only the components readdir and lookup to describe a directory.

This covers the representation of directories in a *Proc* file system, which only leaves the question of how normal files are created. By means of the function proc_read_inode(), the inode for most normal files is assigned the function vector proc_array_inode_operations. All that is implemented in this, however, is the function array_read() in the standard file operations to read the files.

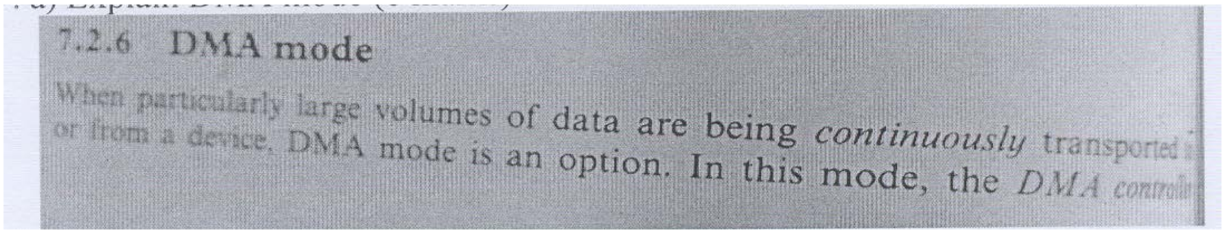
If a process wishes, for example, to read the file /proc/uptime, it allocates a free page of memory to the function array_read() by calling __get_free_page() and passes it to the function get_uptime(). This in turn generates the content of the file by entering the required values in the memory page and returning the size of the buffer (in other words, the file). In the sources, this appears as follows:



```
static int get_uptime(char * buffer)
{
  unsigned long uptime;
  unsigned long idle;

  uptime = jiffies;
  idle = task[0]->utime + task[0]->stime;
#if HZ!=100
  return sprintf(buffer,"%lu.%02lu %lu.%02lu\n",
                uptime / HZ,
                (((uptime % HZ) * 100) / HZ) % 100,
                idle / HZ,
                (((idle % HZ) * 100) / HZ) % 100);

#else
  return sprintf(buffer,"%lu.%02lu %lu.%02lu\n",
                uptime / HZ,
                uptime % HZ,
                idle / HZ,
                idle % HZ);

#endif
}
```

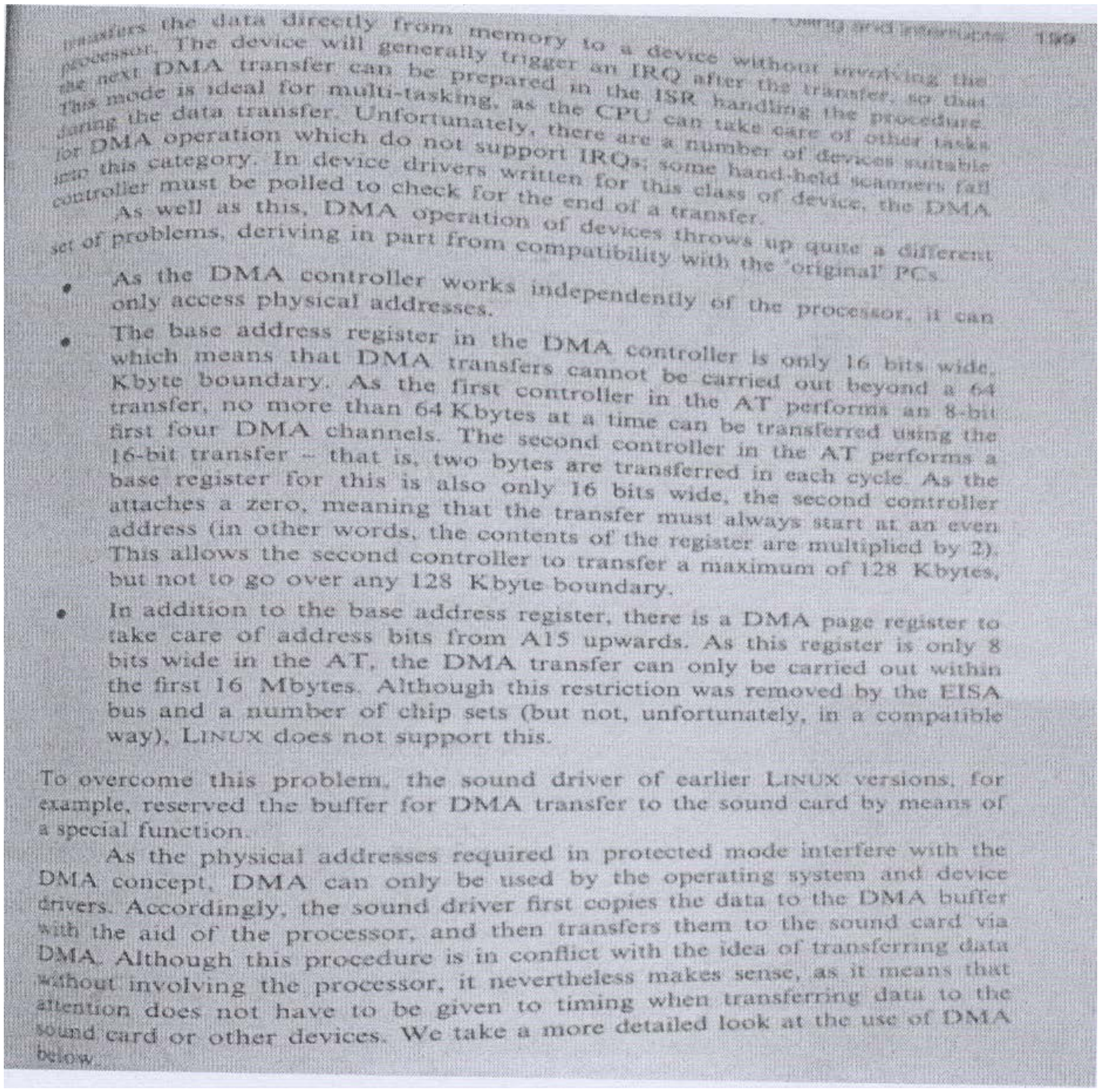
The functions for the individual files are implemented in fs/proc/array.c or in the special sources. The function get_module_list() for the file /proc/module, for example, is located in the file kernel/module.c in the implementation of the

Q.7 a. How do large volumes of data get transported continuously to or from a device? Explain.



7.2.6 DMA mode

When particularly large volumes of data are being *continuously* transported or from a device, DMA mode is an option. In this mode, the *DMA* controll



transfers the data directly from memory to a device without involving the processor. The device will generally trigger an IRQ after the transfer, so that the next DMA transfer can be prepared in the ISR handling the procedure. This mode is ideal for multi-tasking, as the CPU can take care of other tasks during the data transfer. Unfortunately, there are a number of devices suitable for DMA operation which do not support IRQs; some hand-held scanners fall into this category. In device drivers written for this class of device, the DMA controller must be polled to check for the end of a transfer.

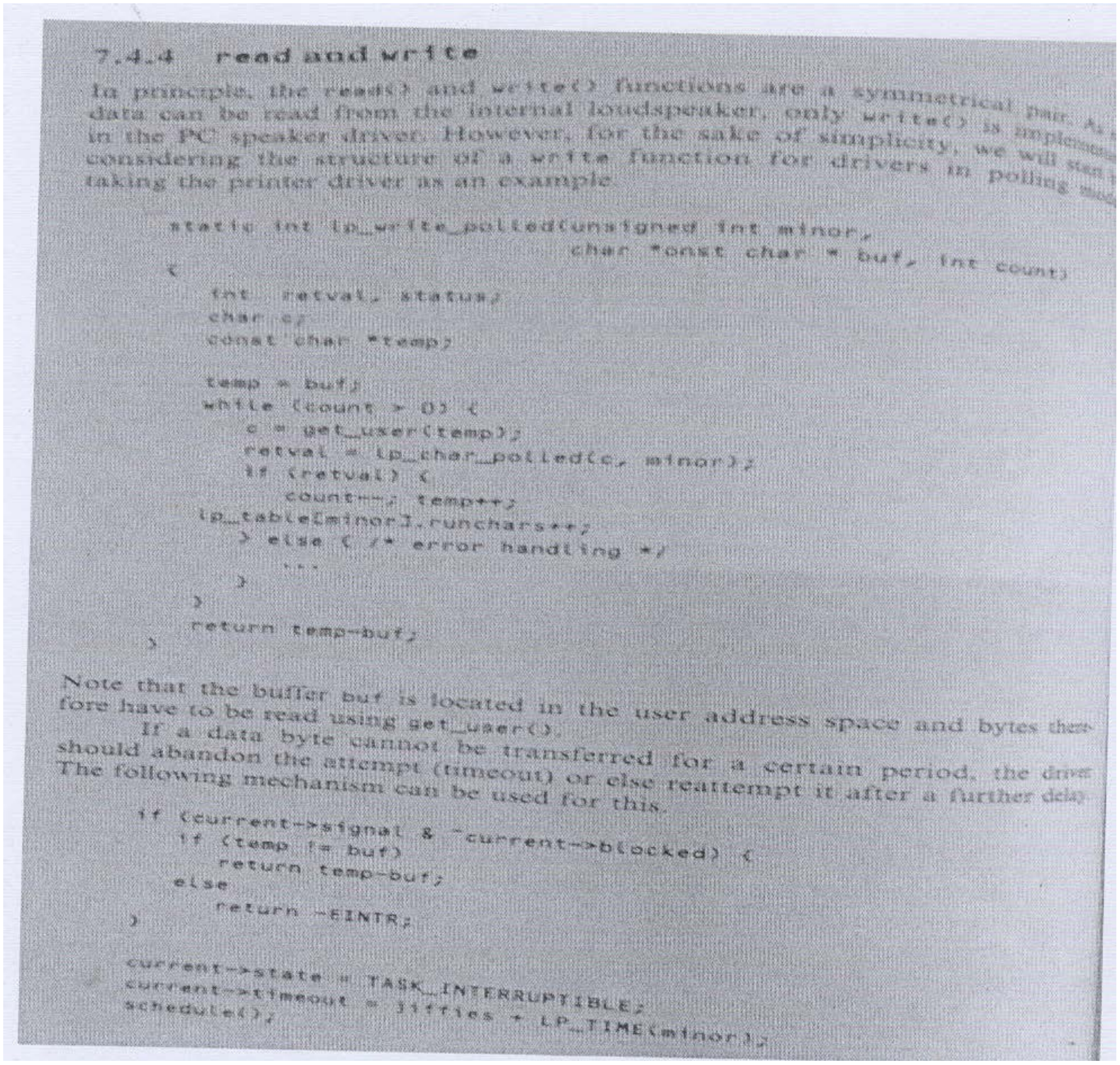
As well as this, DMA operation of devices throws up quite a different set of problems, deriving in part from compatibility with the 'original' PCs.

- As the DMA controller works independently of the processor, it can only access physical addresses.
- The base address register in the DMA controller is only 16 bits wide, which means that DMA transfers cannot be carried out beyond a 64 Kbyte boundary. As the first controller in the AT performs an 8-bit transfer, no more than 64 Kbytes at a time can be transferred using the first four DMA channels. The second controller in the AT performs a 16-bit transfer – that is, two bytes are transferred in each cycle. As the base register for this is also only 16 bits wide, the second controller attaches a zero, meaning that the transfer must always start at an even address (in other words, the contents of the register are multiplied by 2). This allows the second controller to transfer a maximum of 128 Kbytes, but not to go over any 128 Kbyte boundary.
- In addition to the base address register, there is a DMA page register to take care of address bits from A15 upwards. As this register is only 8 bits wide in the AT, the DMA transfer can only be carried out within the first 16 Mbytes. Although this restriction was removed by the EISA bus and a number of chip sets (but not, unfortunately, in a compatible way), LINUX does not support this.

To overcome this problem, the sound driver of earlier LINUX versions, for example, reserved the buffer for DMA transfer to the sound card by means of a special function.

As the physical addresses required in protected mode interfere with the DMA concept, DMA can only be used by the operating system and device drivers. Accordingly, the sound driver first copies the data to the DMA buffer with the aid of the processor, and then transfers them to the sound card via DMA. Although this procedure is in conflict with the idea of transferring data without involving the processor, it nevertheless makes sense, as it means that attention does not have to be given to timing when transferring data to the sound card or other devices. We take a more detailed look at the use of DMA below.

b.Write a note on read() and write() functions.



7.4.4 read and write

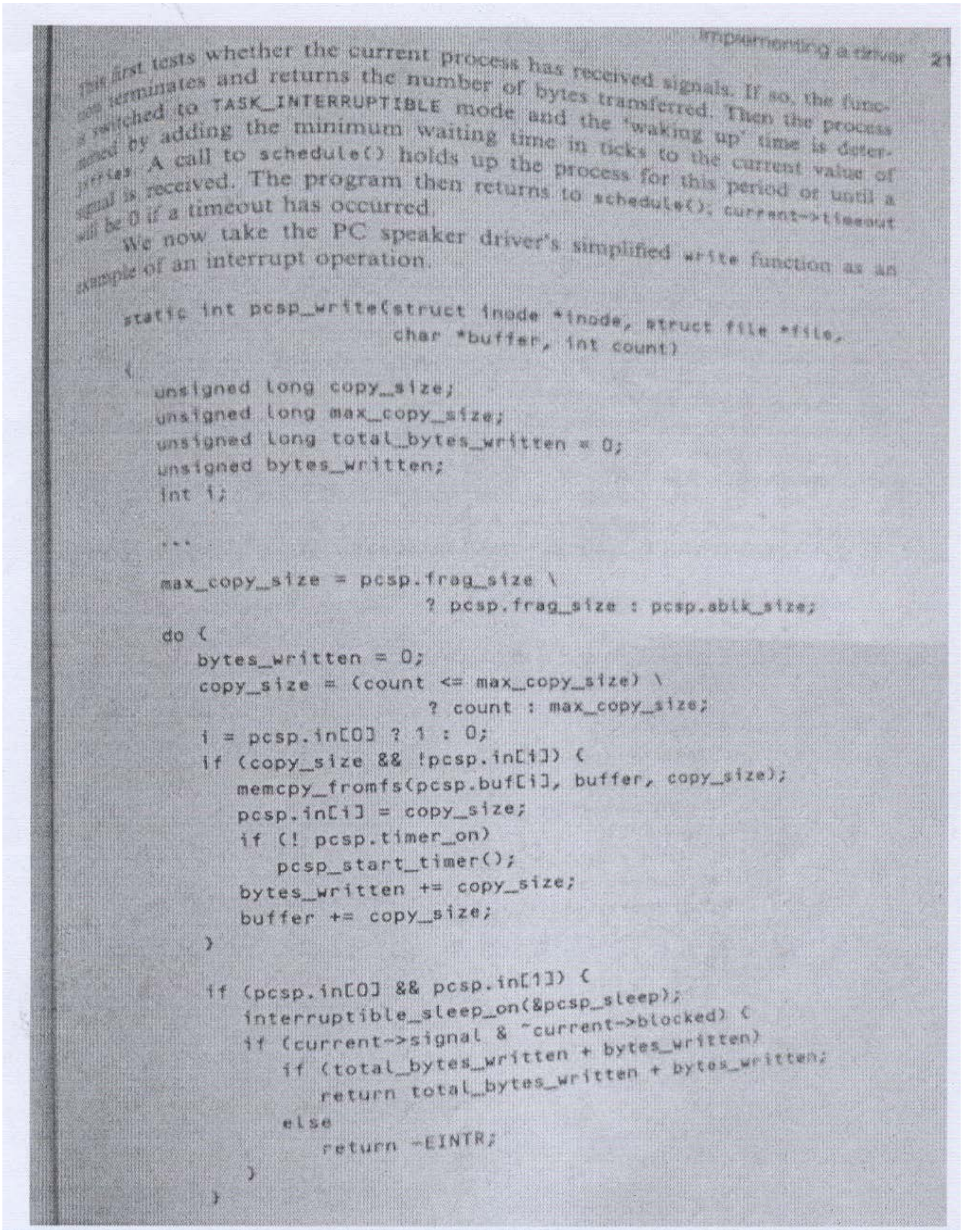
In principle, the read() and write() functions are a symmetrical pair. A[illegible] data can be read from the internal loudspeaker, only write() is impleme[illegible] in the PC speaker driver. However, for the sake of simplicity, we will star[illegible] considering the structure of a write function for drivers in polling mo[illegible] taking the printer driver as an example.

```
static int lp_write_polled(unsigned int minor,
                           char *onst char * buf, int count)
{
    int  retval, status;
    char c;
    const char *temp;

    temp = buf;
    while (count > 0) {
        c = get_user(temp);
        retval = lp_char_polled(c, minor);
        if (retval) {
            count--; temp++;
    lp_table[minor].runchars++;
        } else { /* error handling */
            ...
        }
    }
    return temp-buf;
}
```

Note that the buffer buf is located in the user address space and bytes there-fore have to be read using get_user().

If a data byte cannot be transferred for a certain period, the drive[illegible] should abandon the attempt (timeout) or else reattempt it after a further delay. The following mechanism can be used for this.

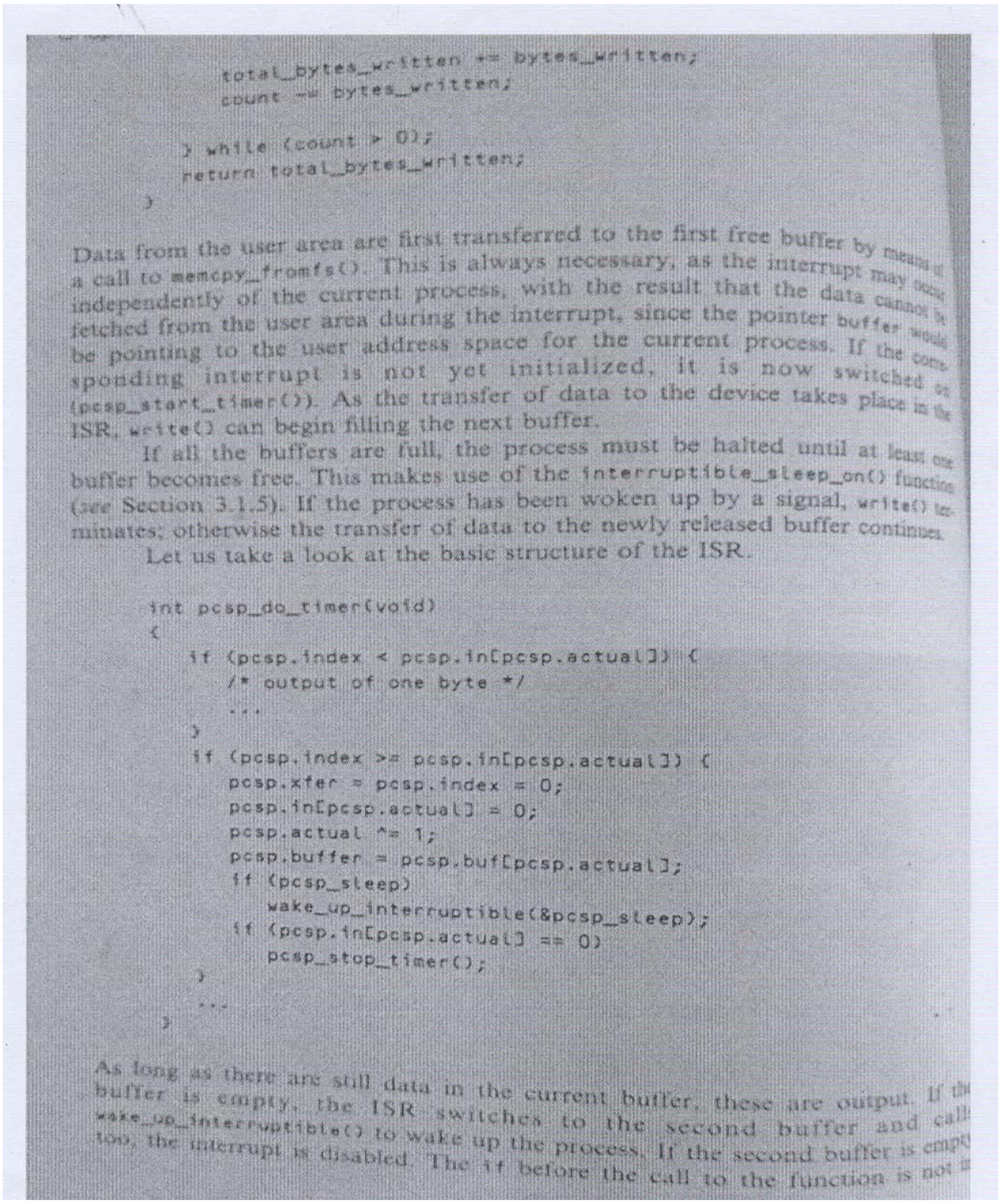
```
if (current->signal & ~current->blocked) {
    if (temp != buf)
        return temp-buf;
    else
        return -EINTR;
}

current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies + LP_TIME(minor);
schedule();
```



This first tests whether the current process has received signals. If so, the function terminates and returns the number of bytes transferred. Then the process is switched to TASK_INTERRUPTIBLE mode and the 'waking up' time is determined by adding the minimum waiting time in ticks to the current value of jiffies. A call to schedule() holds up the process for this period or until a signal is received. The program then returns to schedule(); current->timeout will be 0 if a timeout has occurred.

We now take the PC speaker driver's simplified write function as an example of an interrupt operation.

```
static int pcsp_write(struct inode *inode, struct file *file,
                      char *buffer, int count)
{
    unsigned long copy_size;
    unsigned long max_copy_size;
    unsigned long total_bytes_written = 0;
    unsigned bytes_written;
    int i;

    ...

    max_copy_size = pcsp.frag_size \
                      ? pcsp.frag_size : pcsp.ablk_size;
    do {
        bytes_written = 0;
        copy_size = (count <= max_copy_size) \
                      ? count : max_copy_size;
        i = pcsp.in[0] ? 1 : 0;
        if (copy_size && !pcsp.in[i]) {
            memcpy_fromfs(pcsp.buf[i], buffer, copy_size);
            pcsp.in[i] = copy_size;
            if (! pcsp.timer_on)
                pcsp_start_timer();
            bytes_written += copy_size;
            buffer += copy_size;
        }

        if (pcsp.in[0] && pcsp.in[1]) {
            interruptible_sleep_on(&pcsp_sleep);
            if (current->signal & ~current->blocked) {
                if (total_bytes_written + bytes_written)
                    return total_bytes_written + bytes_written;
                else
                    return -EINTR;
            }
        }
```



```
            total_bytes_written += bytes_written;
            count -= bytes_written;

        } while (count > 0);
        return total_bytes_written;
    }
```

Data from the user area are first transferred to the first free buffer by means of a call to `memcpy_fromfs()`. This is always necessary, as the interrupt may occur independently of the current process, with the result that the data cannot be fetched from the user area during the interrupt, since the pointer `buffer` would be pointing to the user address space for the current process. If the corresponding interrupt is not yet initialized, it is now switched on (`pcsp_start_timer()`). As the transfer of data to the device takes place in the ISR, `write()` can begin filling the next buffer.

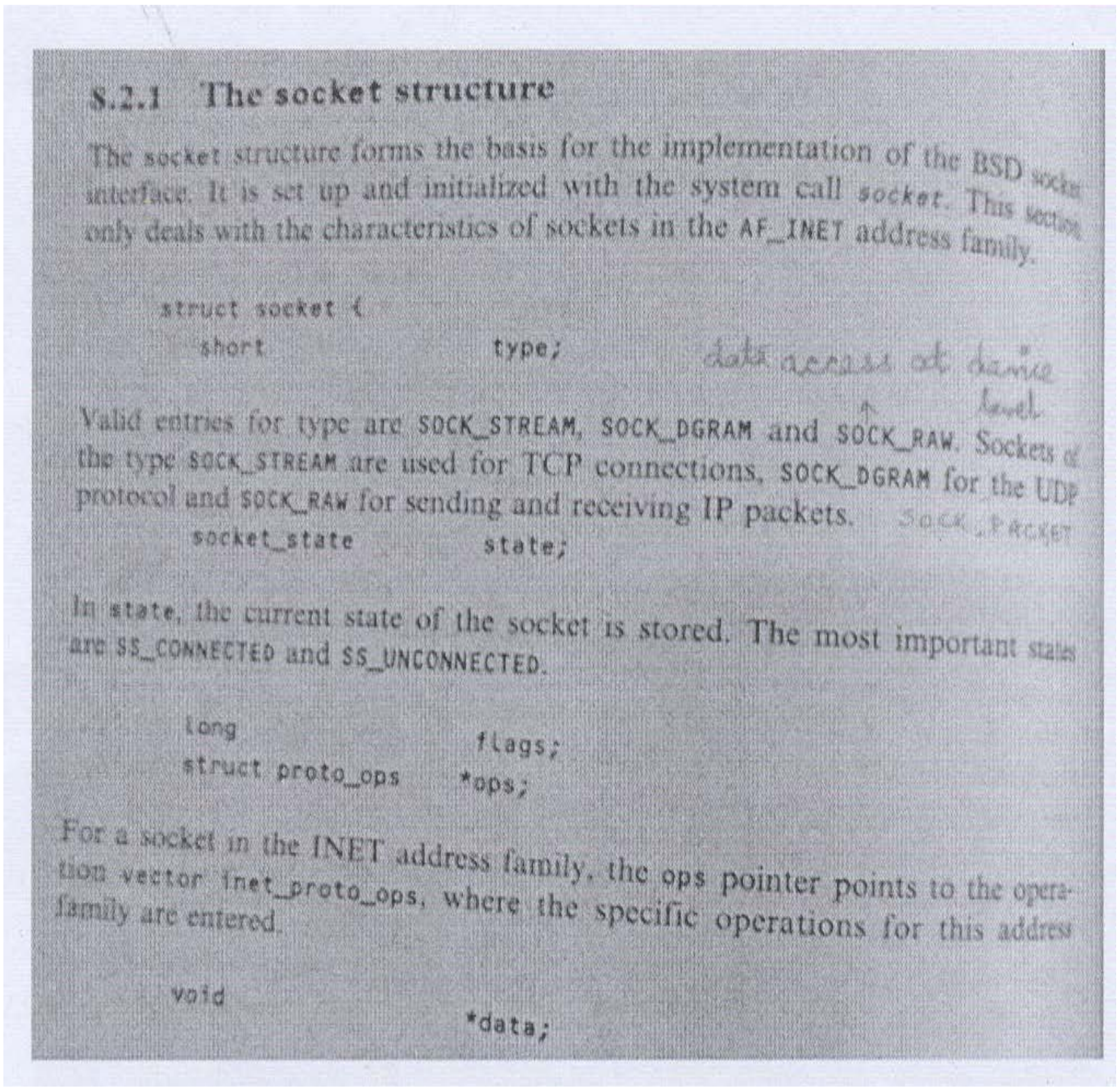
If all the buffers are full, the process must be halted until at least one buffer becomes free. This makes use of the `interruptible_sleep_on()` function (see Section 3.1.5). If the process has been woken up by a signal, `write()` terminates; otherwise the transfer of data to the newly released buffer continues.

Let us take a look at the basic structure of the ISR.

```
    int pcsp_do_timer(void)
    {
        if (pcsp.index < pcsp.in[pcsp.actual]) {
            /* output of one byte */
            ...
        }
        if (pcsp.index >= pcsp.in[pcsp.actual]) {
            pcsp.xfer = pcsp.index = 0;
            pcsp.in[pcsp.actual] = 0;
            pcsp.actual ^= 1;
            pcsp.buffer = pcsp.buf[pcsp.actual];
            if (pcsp_sleep)
                wake_up_interruptible(&pcsp_sleep);
            if (pcsp.in[pcsp.actual] == 0)
                pcsp_stop_timer();
        }
        ...
    }
```

As long as there are still data in the current buffer, these are output. If the buffer is empty, the ISR switches to the second buffer and calls `wake_up_interruptible()` to wake up the process. If the second buffer is empty too, the interrupt is disabled. The `if` before the call to the function is not

Q.8 a. Explain socket structure with the help of a block diagram. Show relationship of the socket with its substructure.



8.2.1 The socket structure

The socket structure forms the basis for the implementation of the BSD socket interface. It is set up and initialized with the system call `socket`. This section only deals with the characteristics of sockets in the `AF_INET` address family.

```
struct socket {
  short                  type;
```

data access at device level

Valid entries for type are `SOCK_STREAM`, `SOCK_DGRAM` and `SOCK_RAW`. Sockets of the type `SOCK_STREAM` are used for TCP connections, `SOCK_DGRAM` for the UDP protocol and `SOCK_RAW` for sending and receiving IP packets. SOCK_PACKET

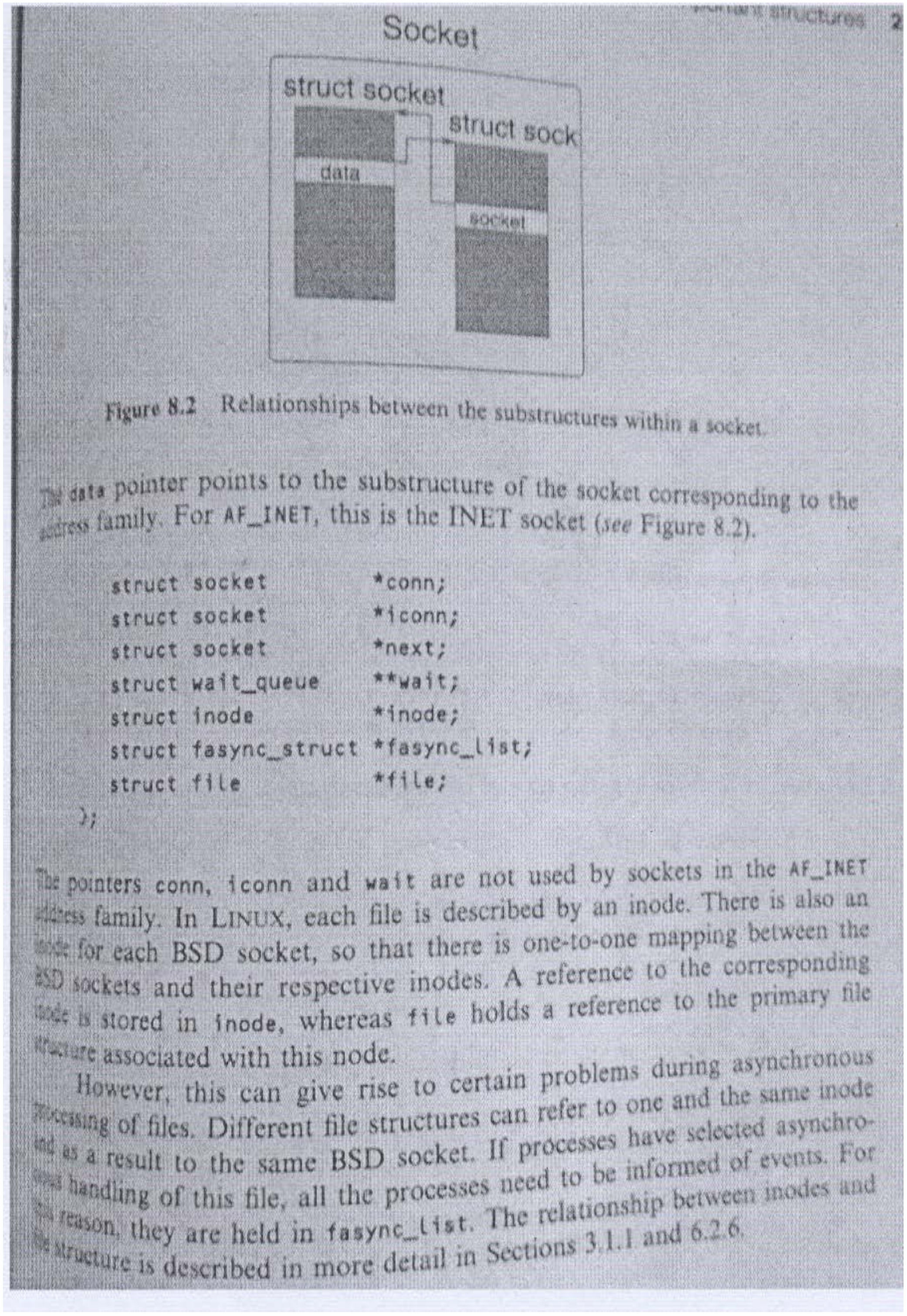
```
  socket_state           state;
```

In `state`, the current state of the socket is stored. The most important states are `SS_CONNECTED` and `SS_UNCONNECTED`.

```
  long                   flags;
  struct proto_ops       *ops;
```

For a socket in the INET address family, the `ops` pointer points to the operation vector `inet_proto_ops`, where the specific operations for this address family are entered.

```
  void
                         *data;
```



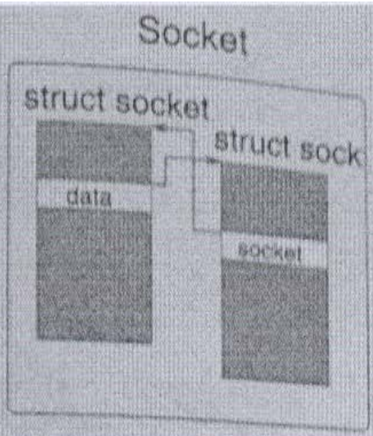


Figure 8.2 Relationships between the substructures within a socket.

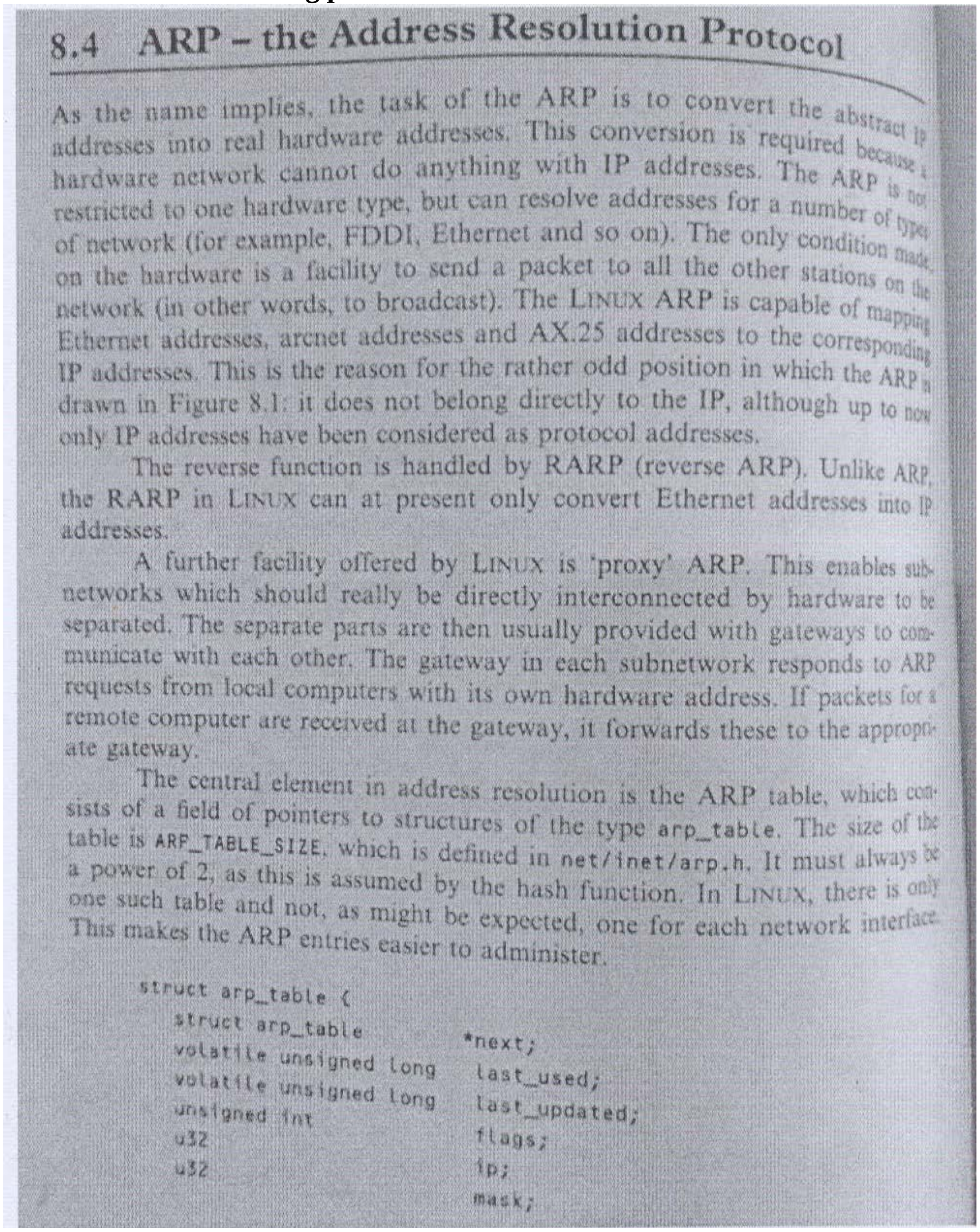
The `data` pointer points to the substructure of the socket corresponding to the address family. For `AF_INET`, this is the INET socket (*see* Figure 8.2).

```
        struct socket          *conn;
        struct socket          *iconn;
        struct socket          *next;
        struct wait_queue      **wait;
        struct inode           *inode;
        struct fasync_struct   *fasync_list;
        struct file            *file;
};
```

The pointers `conn`, `iconn` and `wait` are not used by sockets in the `AF_INET` address family. In LINUX, each file is described by an inode. There is also an inode for each BSD socket, so that there is one-to-one mapping between the BSD sockets and their respective inodes. A reference to the corresponding inode is stored in `inode`, whereas `file` holds a reference to the primary file structure associated with this node.

However, this can give rise to certain problems during asynchronous processing of files. Different file structures can refer to one and the same inode and as a result to the same BSD socket. If processes have selected asynchronous handling of this file, all the processes need to be informed of events. For this reason, they are held in `fasync_list`. The relationship between inodes and file structure is described in more detail in Sections 3.1.1 and 6.2.6.

b.Write notes on following protocols- ARP and IP.



8.4 ARP – the Address Resolution Protocol

As the name implies, the task of the ARP is to convert the abstract IP addresses into real hardware addresses. This conversion is required because a hardware network cannot do anything with IP addresses. The ARP is not restricted to one hardware type, but can resolve addresses for a number of types of network (for example, FDDI, Ethernet and so on). The only condition made on the hardware is a facility to send a packet to all the other stations on the network (in other words, to broadcast). The LINUX ARP is capable of mapping Ethernet addresses, arcnet addresses and AX.25 addresses to the corresponding IP addresses. This is the reason for the rather odd position in which the ARP is drawn in Figure 8.1: it does not belong directly to the IP, although up to now only IP addresses have been considered as protocol addresses.

The reverse function is handled by RARP (reverse ARP). Unlike ARP, the RARP in LINUX can at present only convert Ethernet addresses into IP addresses.

A further facility offered by LINUX is 'proxy' ARP. This enables sub-networks which should really be directly interconnected by hardware to be separated. The separate parts are then usually provided with gateways to communicate with each other. The gateway in each subnetwork responds to ARP requests from local computers with its own hardware address. If packets for a remote computer are received at the gateway, it forwards these to the appropriate gateway.

The central element in address resolution is the ARP table, which consists of a field of pointers to structures of the type `arp_table`. The size of the table is `ARP_TABLE_SIZE`, which is defined in `net/inet/arp.h`. It must always be a power of 2, as this is assumed by the hash function. In LINUX, there is only one such table and not, as might be expected, one for each network interface. This makes the ARP entries easier to administer.

```
struct arp_table {
   struct arp_table              *next;
   volatile unsigned long         last_used;
   volatile unsigned long         last_updated;
   unsigned int                   flags;
   u32                            ip;
   u32                            mask;
```

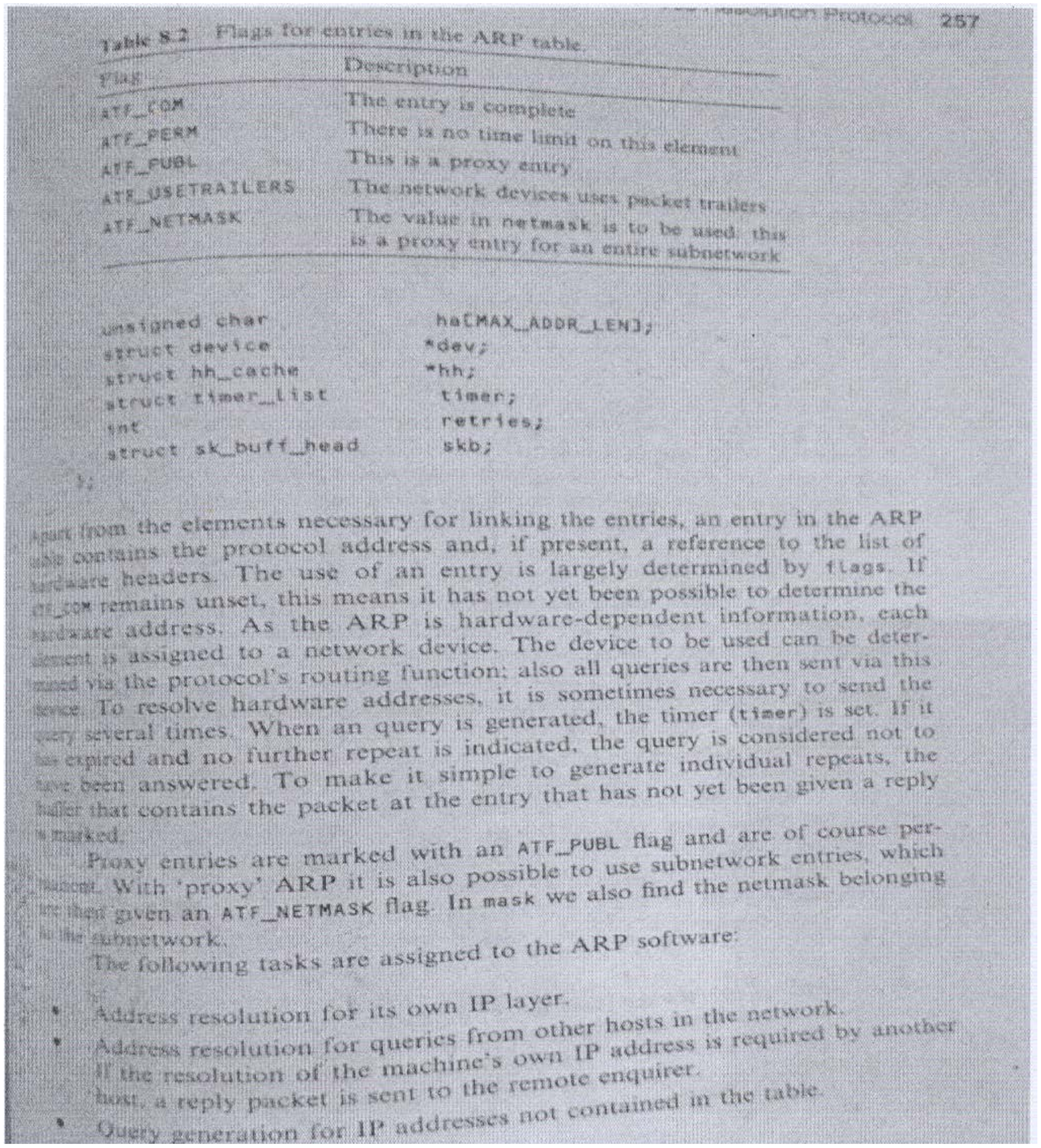


Table 8.2 Flags for entries in the ARP table.

Flag	Description
ATF_COM	The entry is complete
ATF_PERM	There is no time limit on this element
ATF_PUBL	This is a proxy entry
ATF_USETRAILERS	The network devices uses packet trailers
ATF_NETMASK	The value in netmask is to be used: this is a proxy entry for an entire subnetwork

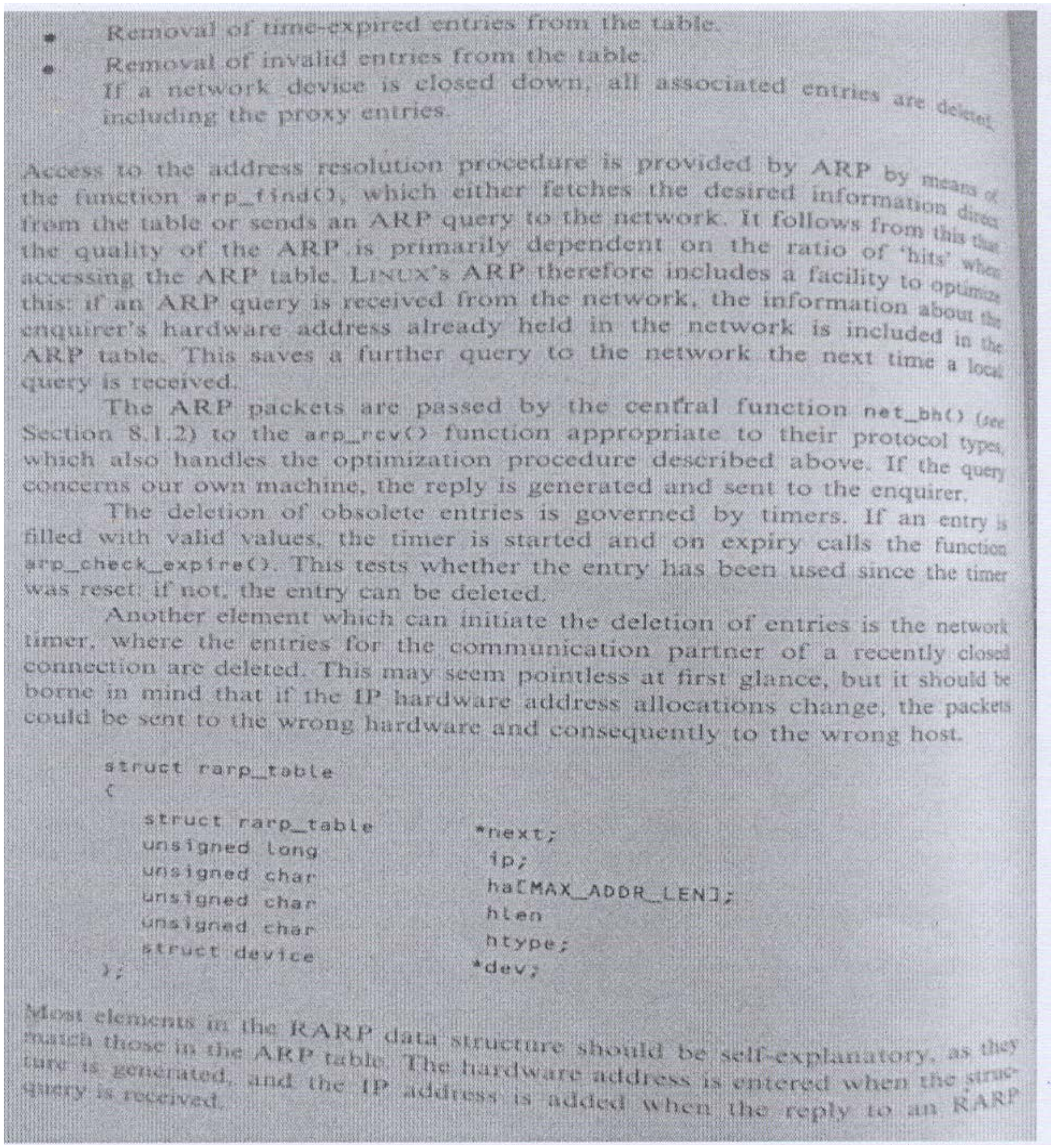
```
unsigned char          ha[MAX_ADDR_LEN];
struct device         *dev;
struct hh_cache       *hh;
struct timer_list      timer;
int                    retries;
struct sk_buff_head    skb;
};
```

Apart from the elements necessary for linking the entries, an entry in the ARP table contains the protocol address and, if present, a reference to the list of hardware headers. The use of an entry is largely determined by flags. If ATF_COM remains unset, this means it has not yet been possible to determine the hardware address. As the ARP is hardware-dependent information, each element is assigned to a network device. The device to be used can be determined via the protocol's routing function; also all queries are then sent via this device. To resolve hardware addresses, it is sometimes necessary to send the query several times. When an query is generated, the timer (timer) is set. If it has expired and no further repeat is indicated, the query is considered not to have been answered. To make it simple to generate individual repeats, the buffer that contains the packet at the entry that has not yet been given a reply is marked.

Proxy entries are marked with an ATF_PUBL flag and are of course permanent. With 'proxy' ARP it is also possible to use subnetwork entries, which are then given an ATF_NETMASK flag. In mask we also find the netmask belonging to the subnetwork.

The following tasks are assigned to the ARP software:

- Address resolution for its own IP layer.
- Address resolution for queries from other hosts in the network. If the resolution of the machine's own IP address is required by another host, a reply packet is sent to the remote enquirer.
- Query generation for IP addresses not contained in the table.



- Removal of time-expired entries from the table.
- Removal of invalid entries from the table.

  If a network device is closed down, all associated entries are deleted, including the proxy entries.

Access to the address resolution procedure is provided by ARP by means of the function `arp_find()`, which either fetches the desired information direct from the table or sends an ARP query to the network. It follows from this that the quality of the ARP is primarily dependent on the ratio of 'hits' when accessing the ARP table. LINUX's ARP therefore includes a facility to optimize this: if an ARP query is received from the network, the information about the enquirer's hardware address already held in the network is included in the ARP table. This saves a further query to the network the next time a local query is received.

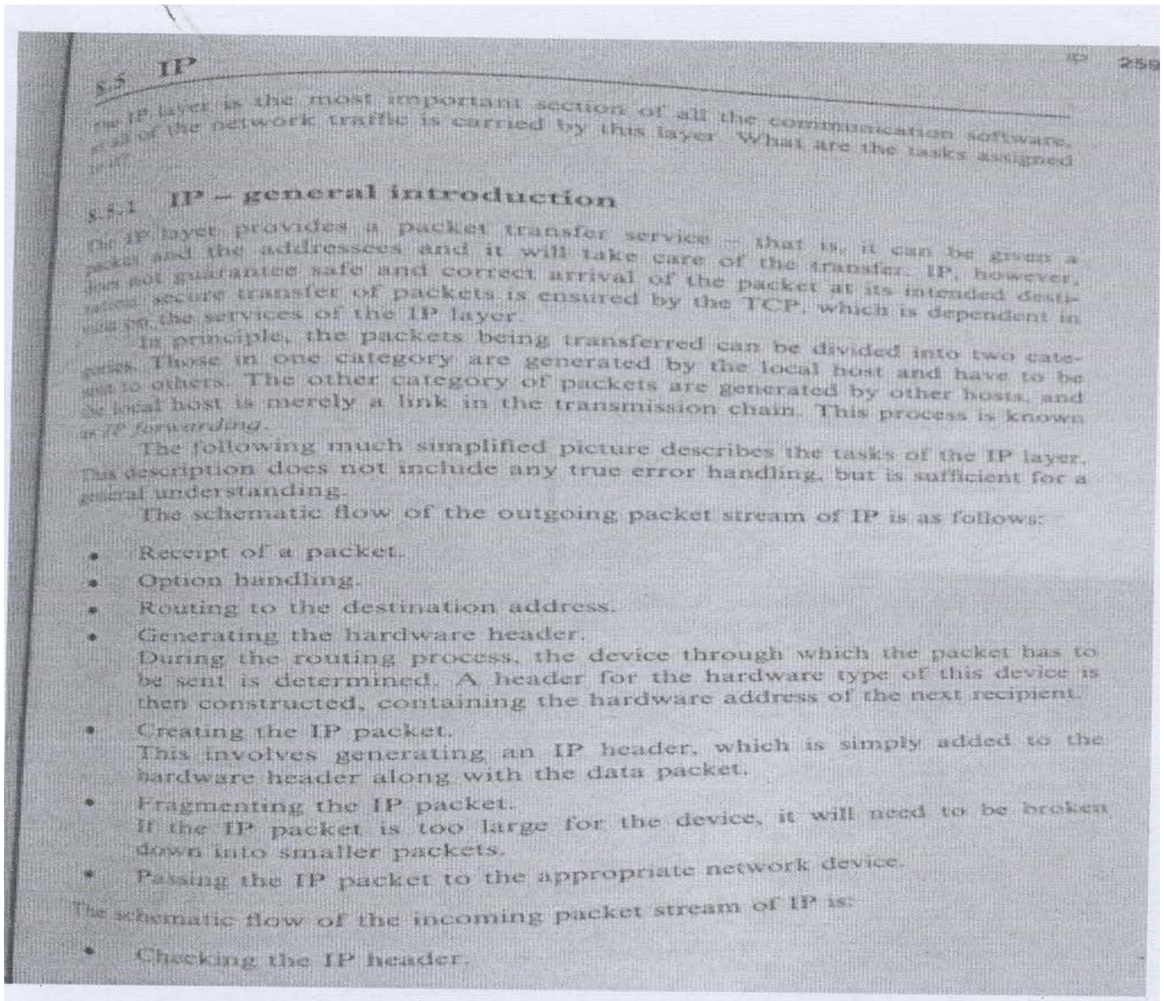
The ARP packets are passed by the central function `net_bh()` (see Section 8.1.2) to the `arp_rcv()` function appropriate to their protocol types, which also handles the optimization procedure described above. If the query concerns our own machine, the reply is generated and sent to the enquirer.

The deletion of obsolete entries is governed by timers. If an entry is filled with valid values, the timer is started and on expiry calls the function `arp_check_expire()`. This tests whether the entry has been used since the timer was reset: if not, the entry can be deleted.

Another element which can initiate the deletion of entries is the network timer, where the entries for the communication partner of a recently closed connection are deleted. This may seem pointless at first glance, but it should be borne in mind that if the IP hardware address allocations change, the packets could be sent to the wrong hardware and consequently to the wrong host.

```
struct rarp_table
{
    struct rarp_table         *next;
    unsigned long              ip;
    unsigned char              ha[MAX_ADDR_LEN];
    unsigned char              hlen
    unsigned char              htype;
    struct device             *dev;
};
```

Most elements in the RARP data structure should be self-explanatory, as they match those in the ARP table. The hardware address is entered when the structure is generated, and the IP address is added when the reply to an RARP query is received.



8.5 IP

The IP layer is the most important section of all the communication software, as all of the network traffic is carried by this layer. What are the tasks assigned to it?

8.5.1 IP – general introduction

The IP layer provides a packet transfer service – that is, it can be given a packet and the addresses and it will take care of the transfer. IP, however, does not guarantee safe and correct arrival of the packet at its intended destination: secure transfer of packets is ensured by the TCP, which is dependent in turn on the services of the IP layer.

In principle, the packets being transferred can be divided into two categories. Those in one category are generated by the local host and have to be sent to others. The other category of packets are generated by other hosts, and the local host is merely a link in the transmission chain. This process is known as *IP forwarding*.

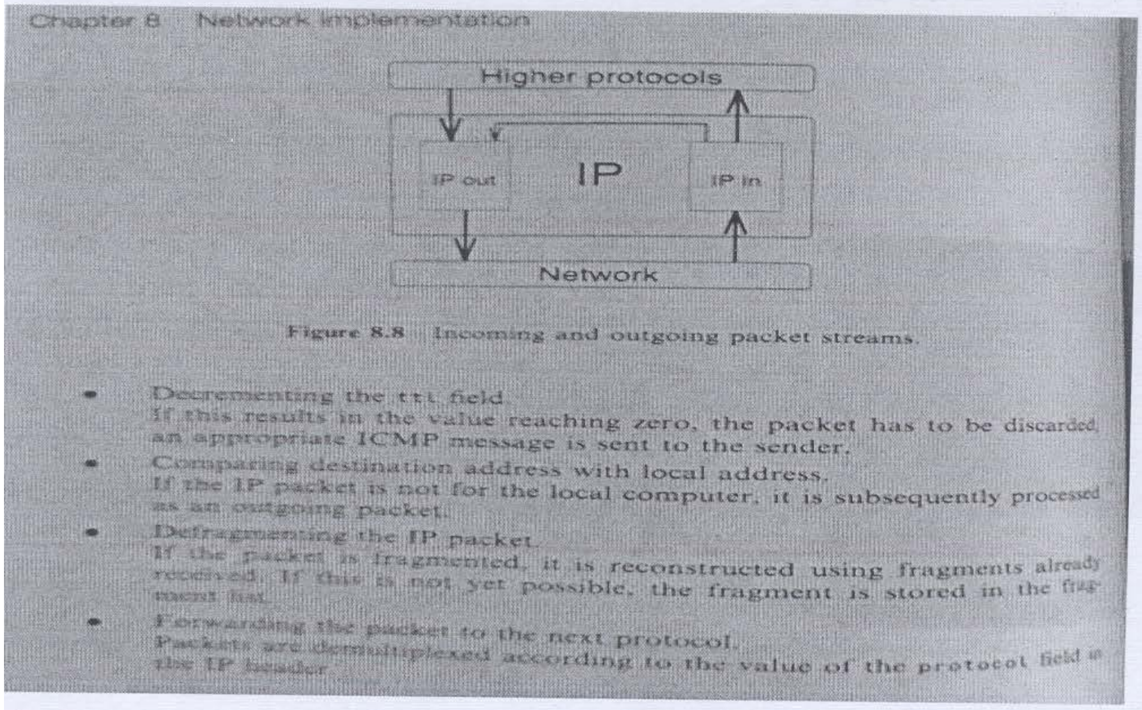
The following much simplified picture describes the tasks of the IP layer. This description does not include any true error handling, but is sufficient for a general understanding.

The schematic flow of the outgoing packet stream of IP is as follows:

- Receipt of a packet.
- Option handling.
- Routing to the destination address.
- Generating the hardware header.
  During the routing process, the device through which the packet has to be sent is determined. A header for the hardware type of this device is then constructed, containing the hardware address of the next recipient.
- Creating the IP packet.
  This involves generating an IP header, which is simply added to the hardware header along with the data packet.
- Fragmenting the IP packet.
  If the IP packet is too large for the device, it will need to be broken down into smaller packets.
- Passing the IP packet to the appropriate network device.

The schematic flow of the incoming packet stream of IP is:

- Checking the IP header.



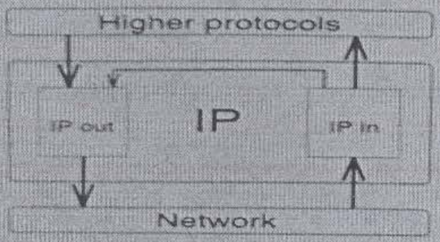
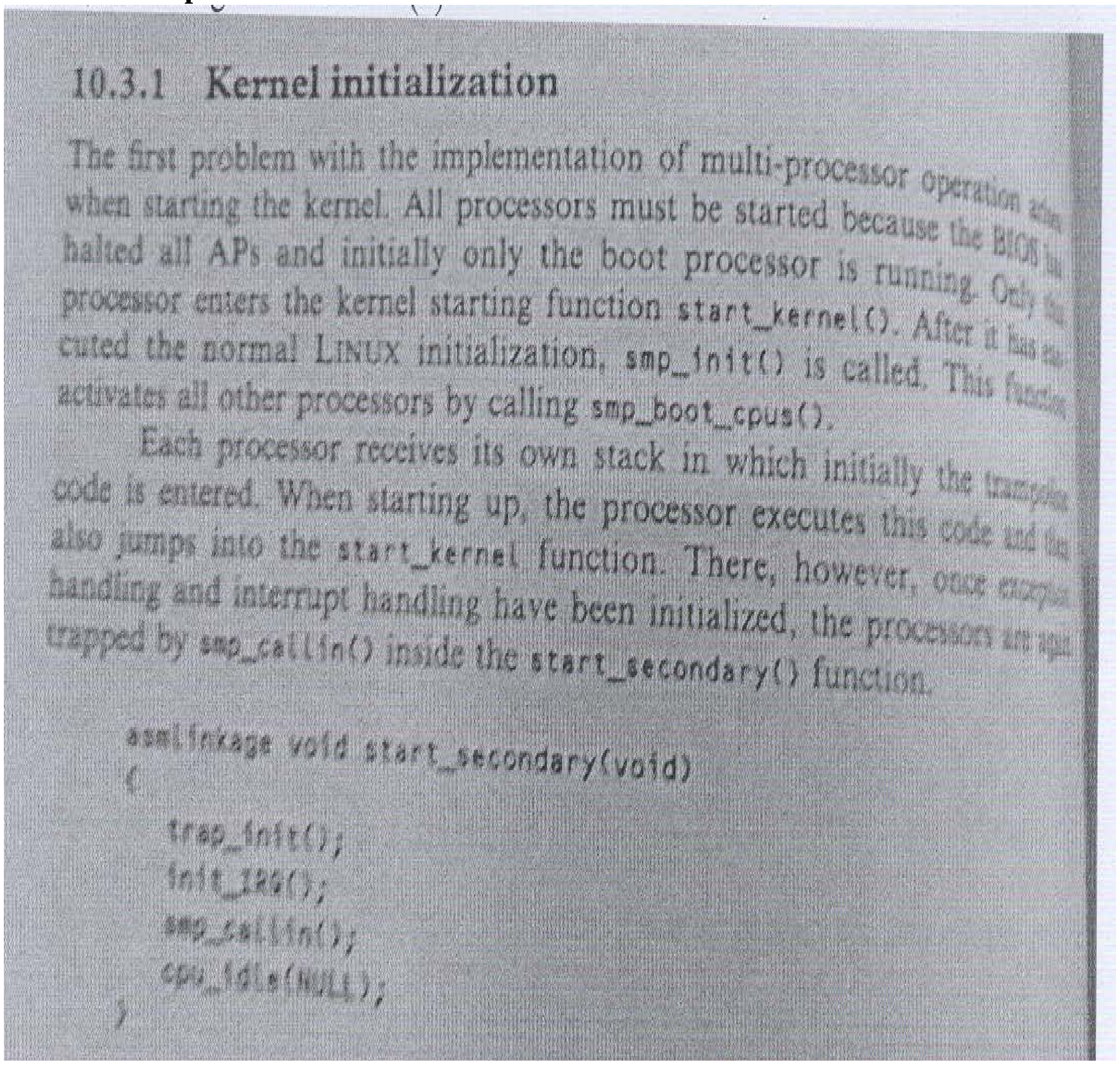


Figure 8.8 Incoming and outgoing packet streams.

- Decrementing the ttl field.
  If this results in the value reaching zero, the packet has to be discarded, an appropriate ICMP message is sent to the sender.
- Comparing destination address with local address.
  If the IP packet is not for the local computer, it is subsequently processed as an outgoing packet.
- Defragmenting the IP packet.
  If the packet is fragmented, it is reconstructed using fragments already received. If this is not yet possible, the fragment is stored in the fragment list.
- Forwarding the packet to the next protocol.
  Packets are demultiplexed according to the value of the protocol field in the IP header.

Q.9 a. In order to implement SMP in Linux kernel, what changes have to be made? Explain.

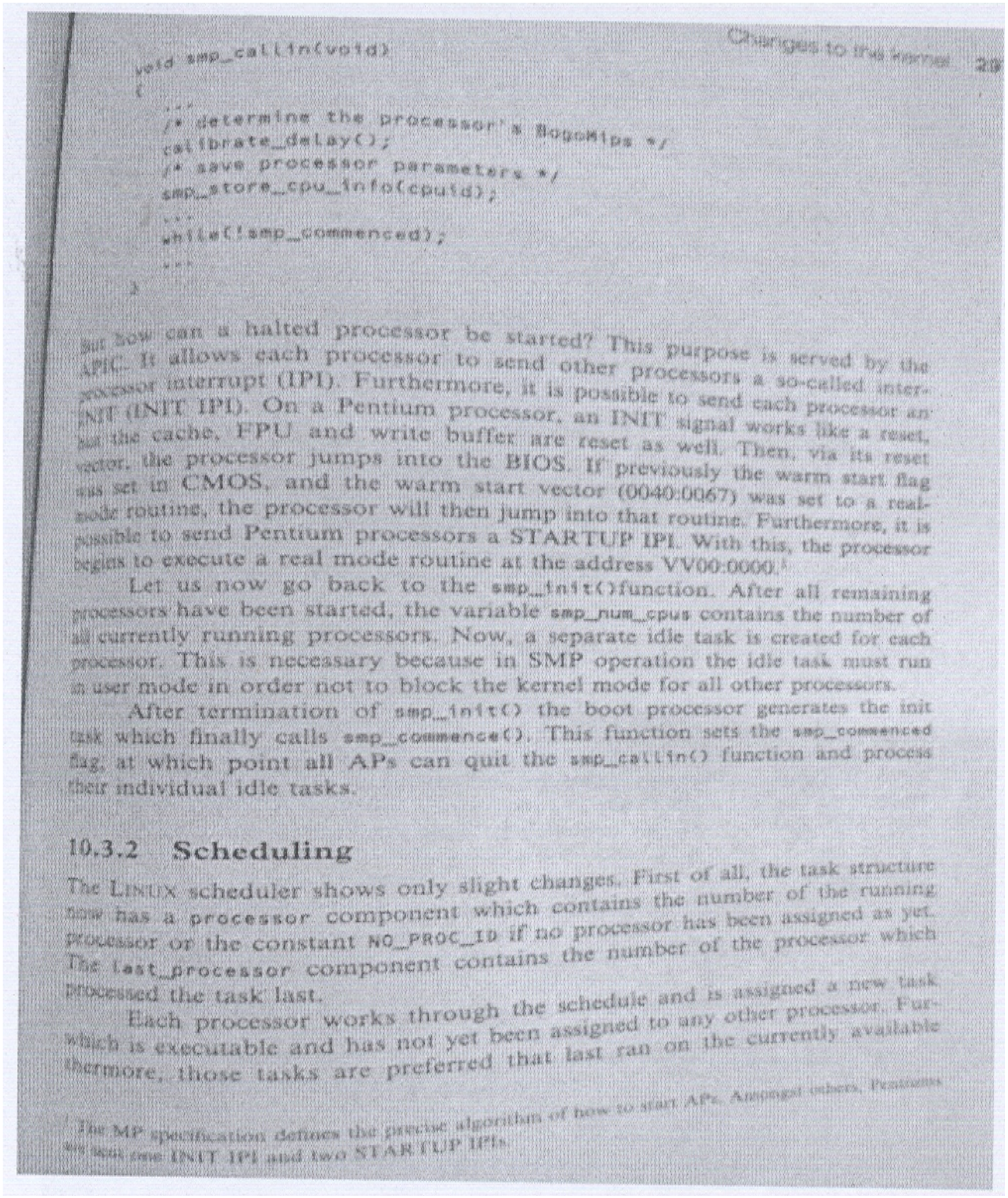


10.3.1 Kernel initialization

The first problem with the implementation of multi-processor operation arises when starting the kernel. All processors must be started because the BIOS has halted all APs and initially only the boot processor is running. Only this processor enters the kernel starting function `start_kernel()`. After it has executed the normal LINUX initialization, `smp_init()` is called. This function activates all other processors by calling `smp_boot_cpus()`.

Each processor receives its own stack in which initially the trampoline code is entered. When starting up, the processor executes this code and then also jumps into the `start_kernel` function. There, however, once exception handling and interrupt handling have been initialized, the processors are again trapped by `smp_callin()` inside the `start_secondary()` function.

```
asmlinkage void start_secondary(void)
{
    trap_init();
    init_IRQ();
    smp_callin();
    cpu_idle(NULL);
}
```



```
void smp_callin(void)
{
    ...
    /* determine the processor's BogoMips */
    calibrate_delay();
    /* save processor parameters */
    smp_store_cpu_info(cpuid);
    ...
    while(!smp_commenced);
    ...
}
```

But how can a halted processor be started? This purpose is served by the APIC. It allows each processor to send other processors a so-called inter-processor interrupt (IPI). Furthermore, it is possible to send each processor an INIT (INIT IPI). On a Pentium processor, an INIT signal works like a reset, but the cache, FPU and write buffer are reset as well. Then, via its reset vector, the processor jumps into the BIOS. If previously the warm start flag was set in CMOS, and the warm start vector (0040:0067) was set to a real-mode routine, the processor will then jump into that routine. Furthermore, it is possible to send Pentium processors a STARTUP IPI. With this, the processor begins to execute a real mode routine at the address VV00:0000.¹

Let us now go back to the `smp_init()`function. After all remaining processors have been started, the variable `smp_num_cpus` contains the number of all currently running processors. Now, a separate idle task is created for each processor. This is necessary because in SMP operation the idle task must run in user mode in order not to block the kernel mode for all other processors.

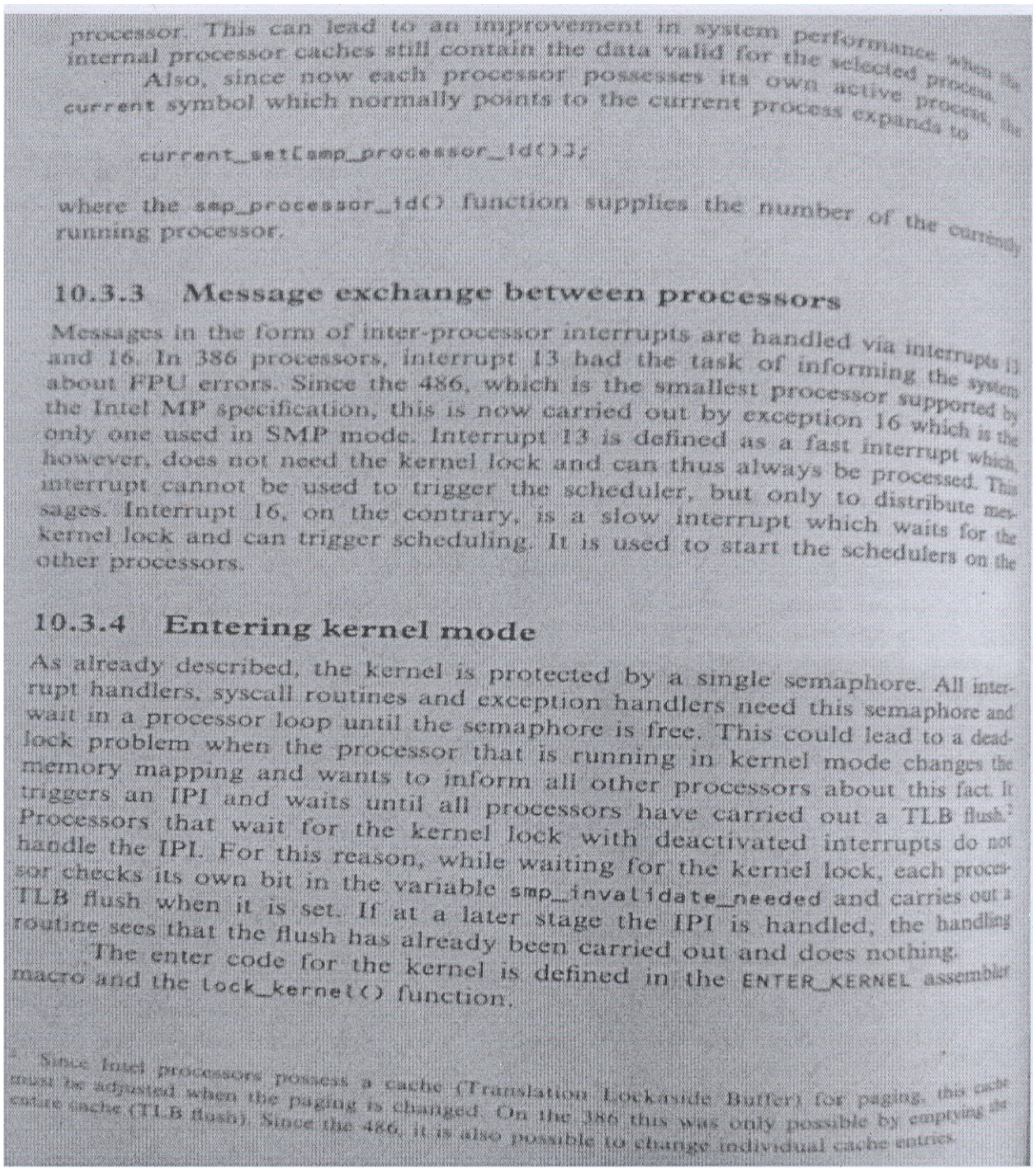
After termination of `smp_init()` the boot processor generates the init task which finally calls `smp_commence()`. This function sets the `smp_commenced` flag, at which point all APs can quit the `smp_callin()` function and process their individual idle tasks.

10.3.2 Scheduling

The LINUX scheduler shows only slight changes. First of all, the task structure now has a `processor` component which contains the number of the running processor or the constant `NO_PROC_ID` if no processor has been assigned as yet. The `last_processor` component contains the number of the processor which processed the task last.

Each processor works through the schedule and is assigned a new task which is executable and has not yet been assigned to any other processor. Furthermore, those tasks are preferred that last ran on the currently available

¹ The MP specification defines the precise algorithm of how to start APs. Amongst others, Pentiums are sent one INIT IPI and two STARTUP IPIs.



processor. This can lead to an improvement in system performance when the internal processor caches still contain the data valid for the selected process.

Also, since now each processor possesses its own active process, the `current` symbol which normally points to the current process expands to

```
current_set[smp_processor_id()];
```

where the `smp_processor_id()` function supplies the number of the currently running processor.

10.3.3 Message exchange between processors

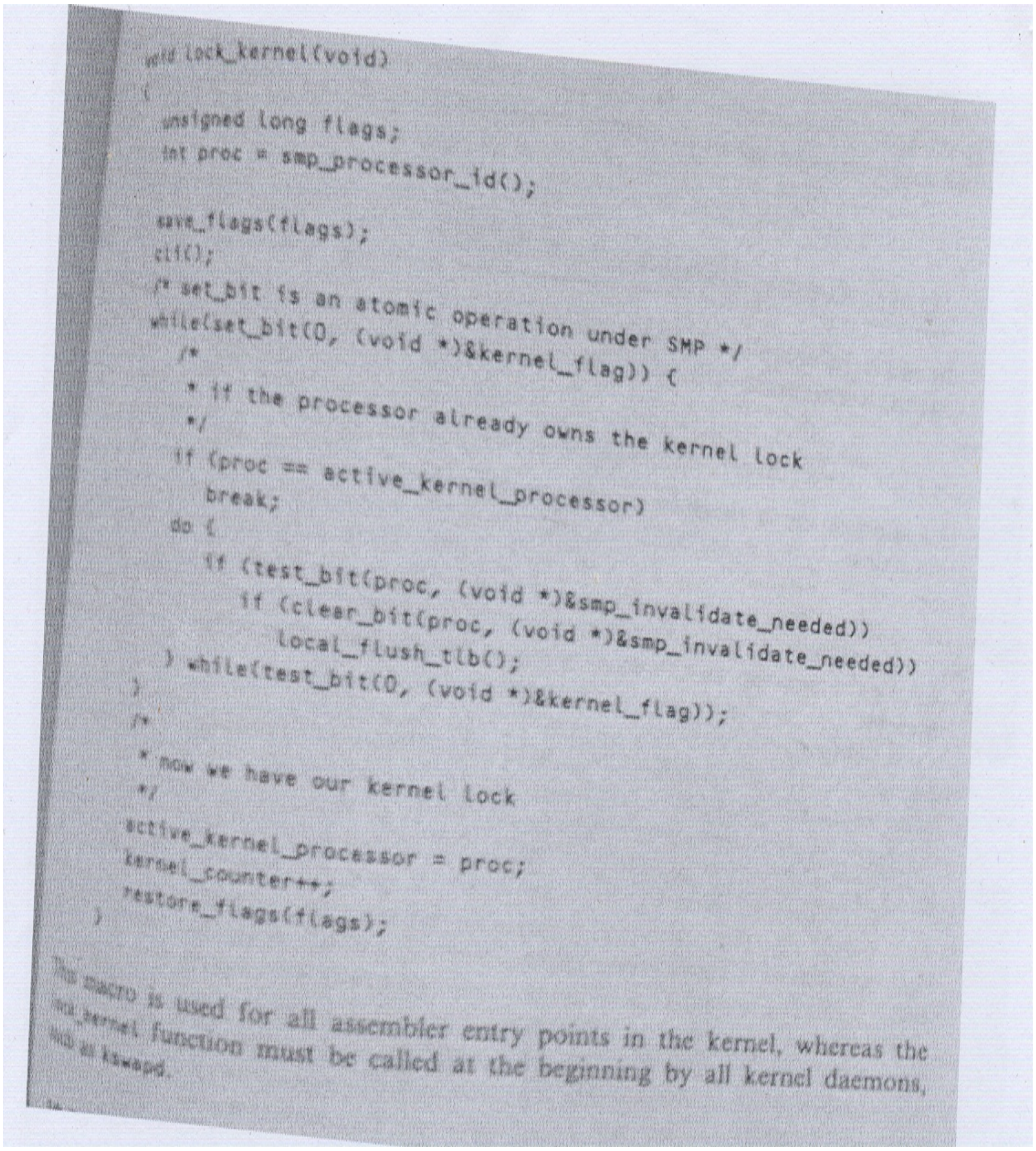
Messages in the form of inter-processor interrupts are handled via interrupts 13 and 16. In 386 processors, interrupt 13 had the task of informing the system about FPU errors. Since the 486, which is the smallest processor supported by the Intel MP specification, this is now carried out by exception 16 which is the only one used in SMP mode. Interrupt 13 is defined as a fast interrupt which, however, does not need the kernel lock and can thus always be processed. This interrupt cannot be used to trigger the scheduler, but only to distribute messages. Interrupt 16, on the contrary, is a slow interrupt which waits for the kernel lock and can trigger scheduling. It is used to start the schedulers on the other processors.

10.3.4 Entering kernel mode

As already described, the kernel is protected by a single semaphore. All interrupt handlers, syscall routines and exception handlers need this semaphore and wait in a processor loop until the semaphore is free. This could lead to a deadlock problem when the processor that is running in kernel mode changes the memory mapping and wants to inform all other processors about this fact. It triggers an IPI and waits until all processors have carried out a TLB flush.[2] Processors that wait for the kernel lock with deactivated interrupts do not handle the IPI. For this reason, while waiting for the kernel lock, each processor checks its own bit in the variable `smp_invalidate_needed` and carries out a TLB flush when it is set. If at a later stage the IPI is handled, the handling routine sees that the flush has already been carried out and does nothing.

The enter code for the kernel is defined in the `ENTER_KERNEL` assembler macro and the `lock_kernel()` function.

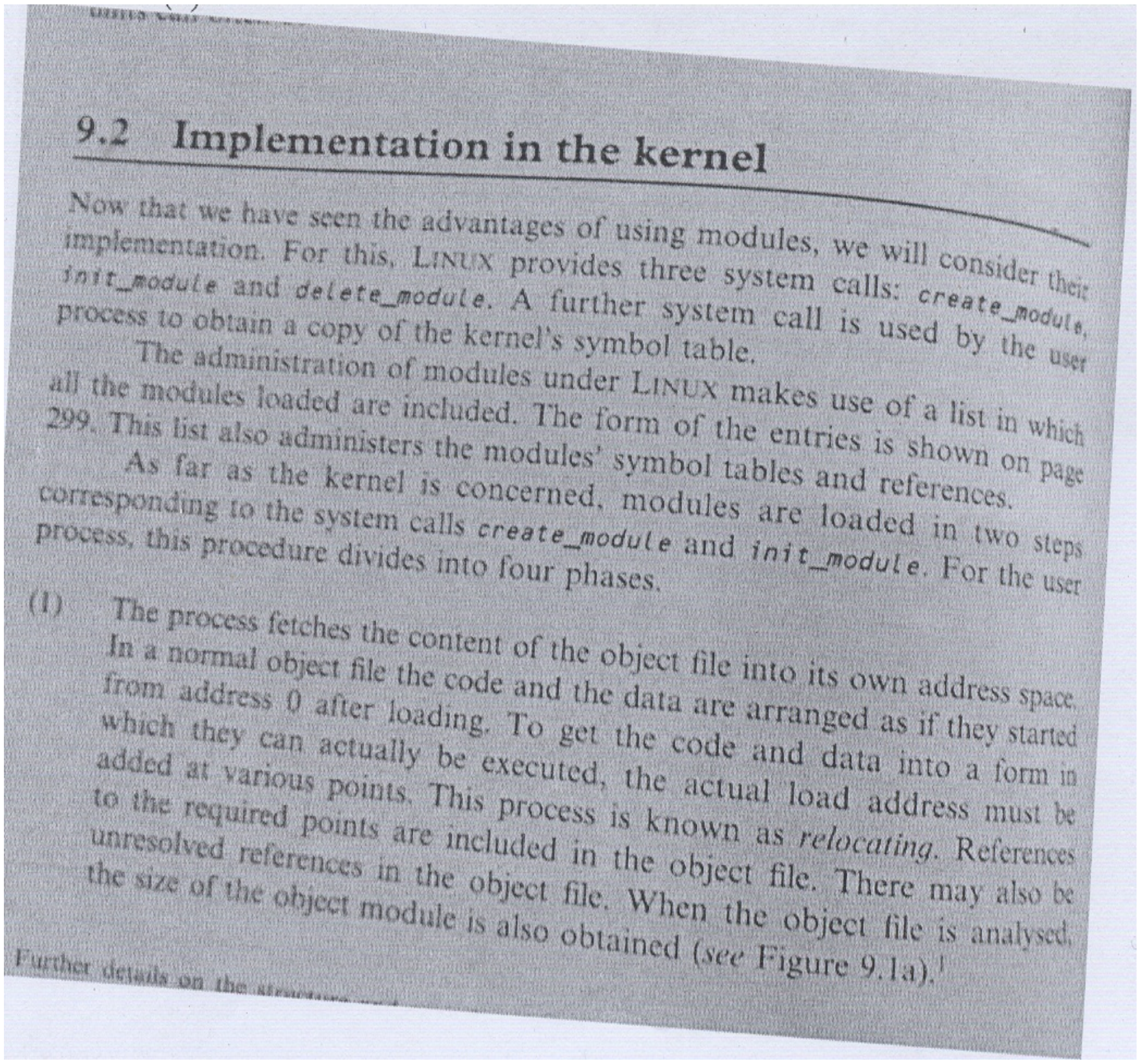
[2] Since Intel processors possess a cache (Translation Lookaside Buffer) for paging, this cache must be adjusted when the paging is changed. On the 386 this was only possible by emptying the entire cache (TLB flush). Since the 486, it is also possible to change individual cache entries.



```
void lock_kernel(void)
{
  unsigned long flags;
  int proc = smp_processor_id();

  save_flags(flags);
  cli();
  /* set_bit is an atomic operation under SMP */
  while(set_bit(0, (void *)&kernel_flag)) {
    /*
     * if the processor already owns the kernel lock
     */
    if (proc == active_kernel_processor)
      break;
    do {
      if (test_bit(proc, (void *)&smp_invalidate_needed))
        if (clear_bit(proc, (void *)&smp_invalidate_needed))
          local_flush_tlb();
    } while(test_bit(0, (void *)&kernel_flag));
  }
  /*
   * now we have our kernel lock
   */
  active_kernel_processor = proc;
  kernel_counter++;
  restore_flags(flags);
}
```

This macro is used for all assembler entry points in the kernel, whereas the lock_kernel function must be called at the beginning by all kernel daemons, such as kswapd.

b.Describe the functions of create_module, init_module and delete_module.



9.2 Implementation in the kernel

Now that we have seen the advantages of using modules, we will consider their implementation. For this, LINUX provides three system calls: `create_module`, `init_module` and `delete_module`. A further system call is used by the user process to obtain a copy of the kernel's symbol table.

The administration of modules under LINUX makes use of a list in which all the modules loaded are included. The form of the entries is shown on page 299. This list also administers the modules' symbol tables and references.

As far as the kernel is concerned, modules are loaded in two steps corresponding to the system calls `create_module` and `init_module`. For the user process, this procedure divides into four phases.

(1) The process fetches the content of the object file into its own address space. In a normal object file the code and the data are arranged as if they started from address 0 after loading. To get the code and data into a form in which they can actually be executed, the actual load address must be added at various points. This process is known as *relocating*. References to the required points are included in the object file. There may also be unresolved references in the object file. When the object file is analysed, the size of the object module is also obtained (*see* Figure 9.1a).[1]

Further details on the

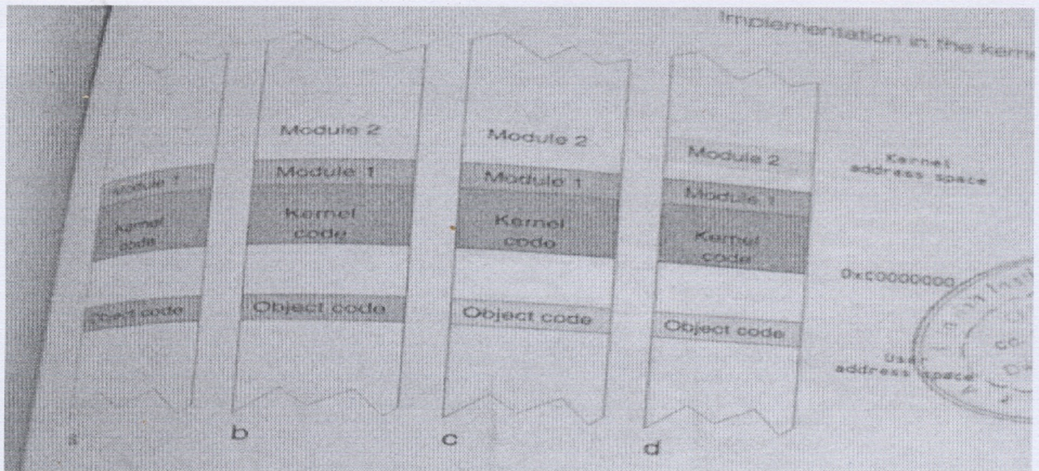


Figure 9.1 The address space on loading a module.

(2) The system call *`create_module`* is now used, firstly to obtain the final address of the object module and secondly to reserve memory for it. To do this, a structure `module` is entered for the module in the list of modules and the memory is allocated. The return value gives us the address to which the module will later be copied (*see* Figure 9.1b).

(3) The load address received by *`create_module`* is used to relocate the object file. This procedure takes place in a memory area belonging to the process – that is, at this point the object module is still not at the right address, but is relocated for the load address of the module in the kernel segment.

Unresolved references can be solved using the kernel symbols, for which LINUX provides the system call *`get_kernel_syms`*. When the function is called, LINUX makes a distinction between two different cases. If the null pointer `NULL` is passed as a parameter, it is possible to find out the size of the kernel's symbol table. If other parameters are used, the location indicated will provide memory for a copy of the symbol table. This enables a process first to determine the table's size and then request a corresponding amount of memory and use the `get_kernel_syms()` system call again. Note that there is no type information of any sort in the table, only addresses. Care must therefore be taken during the development of a module to ensure that the correct header files are included.

To achieve the greatest possible degree of flexibility, the modules themselves can add symbols to the kernel's symbol table. This allows another module to use functions from one loaded earlier. This mechanism is known as *module stacking*. All the symbols exported by a module are collected in a separate symbol table (*see* Figure 9.1c).

(4) Once the preliminary work is complete, we can load the object module. This uses the system call *`init_module`*, which is given among its parameters pointers to a structure `mod_routines` and the module's symbol table. The module's administration functions are entered in the `mod_routines` structure, and LINUX now copies the object module into the kernel address space. The administration function `init()` is called once the code and data have been installed, and within this the relevant register function should also be called.

The return value determines whether or not the installation procedure is judged to have been successful. The second administration function `cleanup()` is called when the module is deinstalled, and initiates the relevant unregister function.

TEXT BOOK

1. Linux Kernel Internals, M. Beck, H. Bome, et al, Pearson Education, Second Edition, 2001