Writing Device Drivers for LynxOS

LynxOS Version 2.5

DOC-0166

Printed in the United States of America.

Contents

Preface

The *Writing Device Drivers for LynxOS* guide describes fundamental concepts of device drivers under LynxOS. The chapters within provide numerous examples to help you develop LynxOS device drivers for your LynxOS applications.

Conventions Used in This Guide

This Preface shows the typographical conventions used in this guide. All references to file names and commands are case sensitive and should be typed exactly as shown.

Fonts

The typefaces in this guide are used to emphasize important concepts. These are described below.

Description	Example
`Courier` font denotes commands, programs, file names, and words that would appear on your computer screen.	`myprog.c` Enter `ls` on the command line
`Bold Courier` font indicates words that you have typed on your computer.	`login:` **`myname`** **`cd /usr/home`**
Words that you must replace with a real name or value appear in *italics*.	`cat` *filename* `mv` *file1 file2*
New terms, book titles, and emphasized words appear in *italics*.	Refer to the *LynxOS Reference Manuals*.

Special Notes

The following notations highlight key points and cautionary notes in this guide.

NOTE: *These callouts note important points in the text.*

CAUTION! *This symbol stresses information that is critical to the integrity or continuity of an operation.*

Technical Support

Lynx Technical Support is available to answer your questions about LynxOS. Please feel free to contact us Monday–Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (Lynx U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Lynx Europe).

In addition, browse our World Wide Web home page for additional information about Lynx products, Frequently Asked Questions (FAQs), and Lynx newsgroups.

World Wide Web

http://www.lynx.com

Lynx U.S. Headquarters

Internet: support@lynx.com
Phone: (408) 879-3940
Fax: (408) 879-3945

Lynx Europe

Internet: tech_europe@lynx.com
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

CHAPTER 1

Driver Basics

A *device driver* can be defined as an interface between the operating system and the hardware device. It is a piece of software designed to hide the idiosyncrasies of a particular device from the kernel. In LynxOS, device drivers are actually a part of the kernel and have complete and unrestricted access to operating system structures. Thus, it is the developer's responsibility to restrict the access of a device driver to only the necessary structures. The mechanism of device driver access by the operating system follows a pre-defined format.

Driver Types

There are actually two types of drivers which can be written under LynxOS:

- Character drivers —Written for devices like serial ports, A/D converters, etc.
- Block device drivers — Written for mass storage devices like disks, ramdisks, tapes, etc. Block device drivers target devices that can be accessed in a "blocked" manner.

There are also network drivers which are considered to be character drivers with special features.

Character Device Drivers

Character drivers are characterized by the following features:

- Arbitrary sized data items—The size or alignment of data which may be transferred through a character device at any point of time is not fixed.
- No kernel buffering—Character device drivers do not utilize kernel buffering. The data from the application is routed directly to the device without being stored in the kernel at any point of time.
- Device is not position-addressable—Character drivers cannot randomly address specific locations on the device. There is no positional information recorded about any characters that have already been processed by the device.

Block Device Drivers

Block device drivers are characterized by the following features:

- Fixed size data — Transfers to and from the device are in a multiple of a fixed size buffer (which in most cases is 512 bytes).
- Kernel buffering — Before a block reaches the block driver, it is actually buffered by the kernel in a “cache”. The end result is that not every block access generated by an application may be sent immediately to the device. It is possible to configure the kernel to turn off this feature for writes by making the cache “write-through”.
- Position of a device (position-addressable) — Unlike character drivers, data given to a block device driver actually has a positional location associated with it. This position is usually established by the

application program. Examples of using positional information generally involve seeking to a particular physical location on the device.

- File system creation — Since block drivers can store data in a randomly-addressable fashion, they can be used to hold a LynxOS file system.

Device Classifications

LynxOS identifies devices using major and minor numbers. A major device corresponds to a separate functional unit that often has its own set of control registers, IO addresses and interrupt vector. A single physical device may be viewed as a number of major devices.

For example, a single data acquisition board consisting of AD and DA converters and clocks could be handled by the driver as three major devices. A minor device identifies sub-units or sub-functions of a major device. Minor devices of the same major device usually share some resources such as control registers and interrupt vector. Examples of minor devices would be partitions on a disk or ports on a multi-port serial board. Minor device numbers also can be used to select different operating modes or functions on a device, for example, a no-rewind option on a tape device.

Referencing Devices and Drivers

Drivers and devices in LynxOS can be referenced using their driver and device identification numbers.

Drivers

Drivers are referenced using a unique driver identification number. This number is assigned automatically during kernel configuration. Drivers supporting both raw(character) and block interfaces have separate driver identification numbers for each interface. The `drivers` command displays the drivers currently installed in the system and their unique driver identification numbers. (See `drivers` in “Device Facilities” section of the *LynxOS Reference Manuals*). Table 1-1 shows a sample output of the `drivers` command.

Table 1-1: : Sample Output of the `drivers` Command

ID	Type	Major Devices	Start	Size	Name
0	char	1	0	0	null
1	char	1	0	0	mem
2	char	1	0	0	ctrl driver
3	char	1	0	0	Raw floppy
4	block	1	0	0	Floppy
5	char	1	0	0	SIM1542 RAW SCSI
6	block	1	0	0	SIM1542 BLK SCSI
7	char	1	0	0	kdconsole
8	char	2	0	0	serial

Devices

Each device is identified by a pair of major/minor numbers. The major numbers are assigned automatically by LynxOS during kernel generation. Character and block interfaces to the same device are indicated by different major numbers.

Major devices installed in the system can be viewed using the `devices` command. (See LynxOS Reference Manual, under “Utility Programs”). The ID column of Table 1-2 gives the major number of the device.

Table 1-2: : Sample Output of the `devices` Command

ID	Type	Driver	Use Count	Start	Size	Name
0	char	0	2	0	0	null device
1	char	1	1	0	0	memory
2	char	2	0	0	0	ctrl dev
3	char	3	0	db0d7008	0	raw Floppy 0-3
4	char	5	1	db0d8a70	0	SIM1542 RAW SCSI
5	char	7	9	db0d8fd8	0	kdconsole
6	char	8	0	db0dc260	0	com 1
7	char	8	0	db0dce40	0	com 2
0	block	4	0	db0d7008	0	Floppy 0-3
1	block	6	2	db0d8a70	0	SIM1542 SCSI

Minor devices are identified by the minor device number. These numbers may be used to indicate devices with different attributes. Minor device numbers are only necessary if there are multiple minor devices per major device. The meaning is selected and interpreted only by the device driver. The kernel does not attach special meaning to the minor number. For example, the driver could use the minor device number to select specific targets, such as a device type, SCSI target ID, or a specific partition.

Application Access to Devices & Drivers

Applications can access devices and drivers using special device files in the file system. These files usually reside in the `/dev` directory (although they can be put anywhere) and are viewable through the `ls -l` command. The device special files are named the same way as regular files and are identified by the device type (character (c) or block (b)) in the first character of the first column of the listing. Special device files do not have any size.

The size column of an `ls -l` listing of such files shows the device’s major and minor numbers, respectively (see Table 1-3 on page 6). Special device files are created with the `mknod` utility.

Table 1-3: : Sample `ls -l` Output of the `/dev` Directory

Permissions	Links	Owner	Major,Minor	Modification Date	Device File Name
crw-rw-rw-	1	root	0,0	Mar 29 01:57	null
crw-r--r--	1	root	1,0	Mar 29 01:52	mem
crw-rw-rw-	1	root	2,0	Mar 29 01:52	tty
crw-rw-rw-	1	root	3,12	Mar 29 01:52	rfd1440.0
crw-------	1	root	4,0	Mar 29 01:52	rsd1542.0
crw-------	1	root	4,16	Mar 29 01:52	rsd1542.0a
crw--w--w-	1	chris	5,0	Mar 29 01:58	atc0
crw--w--w-	1	root	5,1	Mar 29 01:57	atc1
crw-rw-rw-	1	root	6,0	Mar 29 01:52	com1
crw-rw-rw-	1	root	7,0	Mar 29 01:52	com2
brw-rw-rw-	1	root	0,12	Mar 29 01:52	fd1440.0
brw-------	1	root	1,0	Mar 29 01:52	sd1542.0
brw-------	1	root	1,16	Mar 29 01:52	sd1542.0a

Mapping Device Names to Driver Names

The following method can be used to map a device name to a driver:

- Use the `ls -l` command on the `/dev` directory to obtain the listing of all the device names in the system. Determine the major and minor numbers associated with the name of the device. For example, in the above table, the device, `com1`, would be a character device with a major device number of 6 and a minor device number of 0.

- Use the `devices` command to get a listing of all the devices in the system. The `device-id` corresponds to the major device number obtained above. If there is more than one entry with the same device ID, the device type (character or block) eliminates any ambiguity involved. After obtaining the device ID, look in the third column for the driver ID. For example, in Table 1-2 on page 5, `com1` has the driver ID 8.
- Use the `drivers` command to get a listing of all the drivers in the system; with the `driver-id` obtained in the above step, obtain the name of the driver. For `com1`, the driver name is `serial` which is the driver with ID 8 in Table 1-1 on page 4.

CHAPTER 2

Driver Structure

Every device driver consists of a number of entry points and some important data structures. Entry points are functions within the driver that are invoked by the kernel. The important data structures are the *device information structure* and the *statics structure*. The device information structure provides the means for users to configure parameters for each major device.

Device Information Structure

Device Information Definition

The *device information definition* contains the device information necessary to install a major device. This includes information that may vary between instances of a physical device. Examples of information that may change include: physical addresses, interrupt vectors, and available resources. The device information definition may also contain information about configuration defaults. There is one device information definition for every driver.

The implementation of the device information definition is usually done through a C structure which consolidates the information specific to a particular major device. The structure is called *dev*`info` where *dev* corresponds to the device name. For example a printer driver has a device information definition named `ptrinfo etc`. The C structure definition is put in a file typically named *dev*`info.h` where *dev* is a prefix that corresponds to a particular device. The header file may also contain constant definitions. The device information definition files are located in `/sys/dheaders`.

An example of a device information definition for an A/D converter is shown:

```
/* adinfo.h -- Device information definition for A/D
converter */
#define DMA 1
#define NODMA 0
struct adinfo { /* elements aren't optimally ordered */
    short           control_port;
    long            memory_addr;
    unsigned        memory_size;
    short           interrupt_vector;
    char            dma_present;
    unsigned long   default_sampling_rate;
};
```

Device Information Declaration

The device information declaration is an instance of the device information definition. The data is passed to the appropriate driver upon installation. Usually, there is one device information declaration for each major device. Each device information declaration must have a corresponding device information definition.

The device information declaration can be implemented by declaring a C structure in the device information definition file. The device information declaration is named as *dev*`info`*n* where *dev* is the appropriate device name and *n* is a device sequence number, starting at 0. The declaration is put in a C source file named *dev*`info.c` where *dev* stands for the device name. The device information declaration files are located in `/sys/devices`.

The following code segment shows an example of a device information declaration for an A/D converter:

```
/* adinfo.c -- Device information declaration for A/D
   converter */

#include "adinfo.h"
struct adinfo adinfo0 = {
/* elements not optimally ordered */
    /* control_port */              0x100,
    /* memory_addr */               0xA0000,
    /* memory_size */               0x1000,
    /* interrupt_vector */          40,
    /* dma_present */               DMA,
    /* default_sampling_rate */     1000
};
```

Statics Structure

The routines within a driver must communicate with each other and have common access to static information and data structures relating to a particular device such as register addresses, variables indicating the current state of the device, request queues and synchronization semaphores. In a LynxOS driver, this information is held in a C data structure referred to as the *statics* structure. This structure is dynamically allocated and initialized in the install entry point and its address is passed back to the kernel. The kernel then passes this address as the first argument when it calls the other entry points.

Note that there is a separate statics structure for each major device. Therefore, entry points do not need to use the major device number to explicitly identify the current device, its identity is implicit in the address of the statics structure. Most importantly, however, the statics structure is necessary if the user wants to install and uninstall drivers and devices dynamically.

The following is an example of a statics structure:

```
struct ptrstatics {
    int datap;          /* data port address   */
    int controlp;       /* cntrl port address  */
    char control;       /* control bits        */
    int irq;            /* IRQ number          */
    int closing;        /* closing device      */
    int close_sem;      /* semaphore for close */
    int expecting;      /* expecting an int.?  */
    int nextnl;         /* output a '\r' next? */
    int chars;          /* printed since open  */
    int lines;          /* printed since open  */
    int qlen;           /* characters in queue */
    char *q;            /* base queue address  */
    int head;           /* head of queue       */
    int tail;           /* tail of queue       */
    int qdata;          /* data in the queue   */
    int free_sem;       /* free queue space    */
};
```

The format of the statics structure for most drivers can be defined in any way the user wishes. However, for some drivers, namely TTY and network drivers, there are certain rules that must be followed.

Entry Points

A driver consists of a number of *entry point* routines. These routines provide a defined interface with the rest of the kernel and user applications. The kernel calls the appropriate entry point, for example, when a user application makes a system call or when a device interrupt occurs. There is a predefined list of entry points, though a driver does not need to have all the entry points.

A driver may also contain other routines internal to the driver. The names of the entries points follow a conventional pattern, formed by prefixing the entry point name with some character string, usually the name of the driver. Although these are only conventions, they should be followed for consistency and to make drivers more readable.

Install Entry Point

Before a device can be used, a driver must allocate and initialize the necessary data structures and initialize the hardware if it present. This is the responsibility of the `install` entry point in a LynxOS driver.

The install entry point is called multiple times, once for each of the driver's major device that is configured in the system. This is very different from other operating systems where the initialization routine is called only once, irrespective of the number of configured devices. The install entry point receives the address of a user defined device information structure containing the user configurable parameters for the major device. Each invocation of the install entry point receives a different address.

Character Drivers

```
char *devinstall(info)
struct devinfo *info;
{
    struct statics *s;
    int error_found = 0;

    /* Check for existence of device. If present
        allocate and initialise statics struct.  */
    if (device_is_present (info)) {
       s = (struct statics *)sysbrk((long)sizeof(*s));
        bzero (s, sizeof (*s));
        s->io_addr = info->io_addr;
        s->intr_vec = info->intr_vec;
        ...
    } else
        return ((char *)SYSERR);

   /* Initialize device, attach interrupt handler ... */
    error_found = device_init (s);
    if (error_found) {
        sysfree (s, (long) sizeof (*s));
        return ((char *)SYSERR);
    } else
        return ((char *) s);
}
```

In the above example, the functions `device_is_present` and `device_init` would be hardware specific functions internal to the driver.

Block Drivers

```
char *devinstall(info, status)
struct devinfo *info;
struct statics *status;
{
    /* check for existence of device */
    ...
    /* do initialization of the devices */
    /* set interrupt vectors */
    return (char *) status;
}
```

The `install` entry point allocates memory for the statics structure using the `sysbrk` function. Is is important to zero the contents of the memory before use so that all variables in the structure start off in a known state. The variables in the structure should then be initialized. Parameters from the device information structure that are required by other entry points must be copied to the statics structure. The address of the statics structure becomes the return value for the `install` entry point. In the case of an error, the `install` entry point must free the allocated memory using `sysfree` and return `SYSERR`. In this case, the kernel will mark the device as not present and any attempts by an application to open it will receive the error `ENXIO`.

Open Entry Point

The `open` entry point performs the initialization generic to a minor device. For every minor device accessed by the application program, the `open` entry point of the driver is accessed. Thus, if synchronization is required between minor devices (of the same major device), the `open` entry point handles it. Every open system call on a device managed by a driver results in the invocation of the `open` entry point.

Note that the `open` entry point is not re-entrant, though it is pre-emptible. Only one user task can be executing the entry point code at a time for a particular device. Therefore, synchronization between tasks is not necessary in this entry point.

The general structure of the open entry point is shown in the following section.

Character & Block Drivers

```
#include <file.h>
#include <io.h>

int devopen(s, devno, f)
struct statics *s;
int devno;
struct file *f;
{
    /* perform initializing specific to minor device */
     return (OK);
}
```

The variable devno gives the major and minor device number. To obtain the major device number, use the macro major(devno); for the minor device number use the macro minor(devno). The s parameter is the address of the device information structure passed by the kernel. The parameter f is the file structure defined in <file.h>.

The open entry point returns either OK or SYSERR, depending on whether or not the device was successfully opened.

Close Entry Point

The close entry point is only invoked on the "last close" of a device. This means that if multiple tasks have opened the same device node, the close entry point is invoked only when the last of these tasks closes it. Specific allocation done in the open routine is deallocated in the close entry point. As with the open entry point, the close entry point is not re-entrant.

Character Drivers

```
#include <file.h>
#include <io.h>

int devclose(s, f)
struct statics *s;
struct file *f;
{
    /* perform any deallocation done previously in
        open */
    return (OK);
}
```

The `close` entry point returns either `OK` or `SYSERR`, depending on whether or not the device was successfully closed.

Block Drivers

```
#include <file.h>
#include <io.h>

int devclose(s, devno)
struct statics *s;
int devno;
{
    /* perform any deallocation done previously in
        open */
    return (OK);
}
```

Read Entry Point

The `read` entry point copies data from the device into a buffer in the calling task. The structure of the `read` entry point appears as follows:

Character Drivers Only

```
int devread(s, f, buf, count)
struct statics *s;
struct file *f;
char *buf;
int count;
{
    /* do the reading of the bytes from the device */
}
```

The **`s`** parameter is the address of the statics structure passed by the kernel. The **`f`** parameter is the file structure. The parameters **`buf`** and **`count`** refer to the address and size, in bytes, of the buffer where the data should be put. This entry point should return the number of bytes actually read or `SYSERR`.

Write Entry Point

The `write` entry point copies data from a buffer in the calling task to the device. The structure of the `write` entry point appears as follows:

Character Drivers Only

```
int devwrite(s, f, buf, count)
struct statics *s;
struct file *f;
char *buf;
int count;
{
    /* do the write of the bytes to the device */
}
```

The `s` parameter is the address of the statics structure passed by the kernel. The `f` parameter is the file structure. The parameters `buf` and `count` refer to the address and size, in bytes, of the buffer containing the data to be written to the device. This entry point should return the number of bytes actually written or `SYSERR`.

Select Entry Point

The `select` entry point supports I/O polling or multiplexing. The `select` entry point structure would look like this:

Character Drivers

```
select (s, f, which, ffs)
struct statics *s;
struct file *f;
int which;
struct sel *ffs;
{
    switch (which) {
        case SREAD:
            ffs->iosem = &s->n_data;
            ffs->sel_sem = &s->rsel_sem;
            break;
        case SWRITE:
            ffs->iosem = &s->n_spacefree;
            ffs->sel_sem = &s->wsel_sem;
            break;
        case SEXCEPT:
            ffs->iosem = &s->error;
            ffs->sel_sem = &s->esel_sem;
            break;
    }
    return (OK);
}
```

The `which` field is either `SREAD`, `SWRITE`, or `SEXCEPT` indicating that the `select` entry point is monitoring a read, write, or exception condition. The `select` entry point returns `OK` or `SYSERR`.

As well as the `select` entry point code, a number of fields in the statics structure are required to support the select system call:

```
struct statics {
    ...
    int *rsel_sem;      /* sem for select read */
    int *wsel_sem;      /* sem for select write */
    int *esel_sem;      /* sem for select exception */
    int n_spacefree;    /* space available for write */
    int n_data;         /* data available for read */
    int error;          /* error condition */
};
```

The `iosem` field in the `sel` structure is a pointer to a flag (the name `iosem` is misleading, it is not a semaphore) that indicates whether the condition being polled by the user task is true or not. The `sel_sem` field is a pointer to a semaphore that the driver signals at the appropriate time (see below). The value of the semaphore itself is managed by the kernel and should *never* be modified by the driver. A driver must always set the `iosem` and `sel_sem` fields in the `select` entry point.

A driver that supports select must also test and signal the select semaphores at the appropriate points in the driver, usually the interrupt handler or kernel thread. This should be done when data becomes available for reading, when space is available for writing or when an error condition is detected:

```
/* data input */

s->n_data++;
disable (ps);
if (s->rsel_sem)
    ssignal (s->rsel_sem);
restore (ps);

/* data output */

s->n_spacefree++;
disable (ps);
if (s->wsel_sem)
    ssignal (s->wsel_sem);
restore (ps);

/* errors, exceptions */

if (error_found) {
    s->error++;
    disable (ps);
    if (s->esel_sem)
        ssignal (s->esel_sem);
    restore (ps);
}
```

Ioctl Entry Point

The `ioctl` entry point is called when the `ioctl` system call is invoked for a particular device. This entry point is used to set certain parameters in the device or obtain information about the state of the device. The `ioctl` entry point’s format looks like this:

Character Drivers

```
int devioctl(s, f, command, arg)
struct statics *s;
struct file *f;
int command;
char *arg;
{
    /* depending on the command, copy relevant
        information to or from the arg structure */
}
```

Block Drivers

```
int devioctl(s, devno, command, arg)
struct statics *s;
int devno;
int command;
char *arg;
{
    /* depending on the command copy relevant
        information to or from the arg structure */
}
```

The driver defines the meaning of `command` and `arg` except for `FIOPRIO` and `FIOASYNC`, which are used by LynxOS to communicate with the drivers. If the `arg` field is to be used as a memory pointer, it should first be checked for validity with either `rbounds` or `wbounds`. (See "Driver Service Calls" in the *LynxOS Reference Manuals*).

The kernel uses `FIOPRIO` to signal the change of a task priority to the driver that is doing priority tracking. `FIOASYNC` is invoked when a task invokes the `fcntl()` system call on an open file, setting or changing the value of the `FNDELAY` or `FASYNC` flag.

This entry point should return `OK` or `SYSERR`.

Uninstall Entry Point

The `uninstall` entry point is called when a major device is uninstalled dynamically from the system (via the `devinstall` command or `cdv_uninstall` and `bdv_uninstall` system calls). It is not called by a normal reboot of the system.

Interrupt vectors set in the `install` routine and dynamically-allocated data is freed in the `uninstall` entry point. The entry point should also perform any necessary actions to put the hardware device into an inactive state. Once the interrupt handler is detached, any further interrupts from the device would cause the system to crash. The format of the `uninstall` entry point is shown below.

Character Drivers

```
void devuninstall(s)
struct statics *s;
{
    ...
    /* clear interrupt vectors, disable device etc */
    ...
    sysfree(s, (long)sizeof(*s));
    ...
}
```

Block Drivers

```
void devuninstall(info)
struct devinfo *info;
{
    ...
    /* clear interrupt vectors */
    ...
    ...
}
```

The statics structure is freed in the `uninstall` entry point using the call `sysfree()`. See “Driver Service Calls” in the *LynxOS Reference Manuals*. The interrupt vectors are cleared here using `iointclr()`. The return value is irrelevant.

Strategy Entry Point

The `strategy` entry point is valid only for block devices. Instead of a `read` and `write` entry point, block device drivers have a `strategy` routine which handles both read and write. The format of the `strategy` entry routine is as follows:

```
#include <disk.h>
devstrategy(status, bp)
struct statics *status;
struct buf_entry *bp;
{
    /* do read or write depending on whether a read
   or write operation is specified in the buf_entry */
}
```

The structure `buf_entry` is defined in `<disk.h>` and is reproduced below.

```
struct buf_entry {
    struct buf_entry *b_forw;   /* double link list for
owner device */
    struct buf_entry *b_back;
    struct buf_entry *av_forw;/* free list pointers*/
    struct buf_entry *av_back;/*can be used by driver*/
        char *memblk;   /* data block */
        int w_count;
        int rw_count;
        int b_rwsem;/*block read/write semaphore*/
        int b_error;/* disk error */
        int b_sem;  /* block semaphore */
        int b_status;/* block status */
        int b_device;/* block device number */
        long b_number;/* block number */
};
```

The size of the block request is returned by the macro `BLKSIZE(b_device)`. `BLKSIZE` takes as its argument the block device number, `b_device`.

Every block to be read or written has an associated buffer entry. The encoded major and minor device numbers are stored in `b_device`. The memory source or destination is denoted by `memblk`. The source or destination number of the block on the device is `b_number`. The `b_status` is used to indicate the status and type of transfer. If the value `B_READ` is set, the transfer is a read; otherwise, the transfer is a write.

The transfer status is indicated by `B_DONE` or `B_ERROR`. The `b_error` field is set non-zero if the transfer fails. The `b_rwsem` semaphore signals once the transfer completes. `av_forw` points to the next buffer or `NULL` if the end of the list has been reached. w_count is available to the driver for its own purposes. For example, LynxOS drivers use w_count for priority tracking.

CHAPTER 3

Memory Management

LynxOS Virtual Memory Model

LynxOS uses a *virtual memory* architecture. All memory addresses generated by the CPU are translated by the hardware MMU (memory management unit). Each user task has its own protected virtual address space which prevents tasks from inadvertently (or maliciously) interfering with each other. The kernel (which includes device drivers) and the currently executing user task exist within the same virtual address space—the user task is mapped into the lower part, the kernel into the upper part. Only the part of the virtual map occupied by the user task is remapped during a context switch.

Applications cannot access the part of the address space occupied by the kernel and its data structures. The constant `OSBASE` defines the limit of the user accessible space. Addresses above this limit are accessible only by the kernel.

Kernel code, on the other hand, has access to the entire virtual address space. This greatly facilitates the passing of data between drivers and user tasks. A device driver, as part of the kernel, can read or write a user address as a direct memory reference, without the need to use special functions. However, this also means that it is your responsibility as developer to restrict the access of your device driver to only the necessary structures.

Please see Chapter 5, “Accessing Hardware” for detailed LynxOS virtual memory maps of the currently supported platforms.

LynxOS Address Types

A LynxOS driver has to deal with several different address types. They include:

- user virtual
- kernel virtual
- kernel direct mapped
- physical

User Virtual

These are addresses passed to the driver from a user application, typically addresses of buffers or data structures used to transfer data or information between the application and a device or the driver. They are only valid when the user task that passed the address is the current task i.e. in a top-half entry point.

Kernel Virtual

The addresses of kernel functions and variables that can be used by a driver. The mapping of kernel virtual addresses is permanent so they are valid from anywhere within a driver. They are not accessible from an application.

Kernel Direct Mapped

A region of the kernel virtual space that is directly mapped to physical memory (i.e. contiguous virtual addresses map to contiguous physical addresses). The base of this region is defined by the constant `PHYSBASE` which maps to the start of RAM. The size of the region is platform dependent.

Physical

Drivers never reference physical addresses directly, but they are sometimes required by devices such as DMA controllers that do not use the MMU.

Allocating Memory

The main requirements for dynamic memory allocation in a driver are:

- the statics structure
- data buffers, queues

Memory that is used throughout the life of the driver should be allocated using `sysbrk`. The most common example is the statics structure which is allocated and initialized in the *install* entry point. This memory remains allocated until the driver frees it when the major device is uninstalled. Memory allocated with `sysbrk` is freed using `sysfree`, which takes as arguments the address returned by sysbrk and the number of bytes.

```
char *
drvinstall (info)
struct devinfo *info;
{
    struct statics *s;

    /* allocate statics structure */
   s = (struct statics *) sysbrk ((long) sizeof (*s));
    if (s == NULL)
        return (SYSERR);
   ....                   /* initialize h/w and s/w */
    return ((char *) s);   /* pass statics address
                                back to kernel */
}

drvuninstall (s)
struct statics *s;
{
    ...
    /* free statics memory */
    sysfree (s, (long) sizeof (*s));
    return (OK);
}
```

The size argument passed to sysbrk and sysfree is rounded up to a multiple of 8 bytes. Therefore, the address returned by sysbrk will always be aligned on a long word (8 byte) boundary.

The functions sysbrk/sysfree implement a kernel memory heap. Memory allocated with sysbrk is mapped into the kernel's data segment. When memory is freed with sysfree it remains allocated to the kernel and can only be reused by subsequent calls to sysbrk. So these functions should not be used for temporary storage needs.

Memory can also be allocated using the kernel function get1page. As the name suggests this allocates one page (PAGESIZE bytes, typically 4 KB) of memory. Unlike sysbrk, get1page does not map memory in the kernel's data segment. Rather it returns the kernel direct mapped address of the memory page. When this memory is freed using free1page, the memory is returned to the system for general use. This makes get1page/free1page suitable for temporary memory requirements. Another important feature of get1page is that the returned address is always aligned on a page boundary. This makes it particularly convenient for allocating buffers for use in DMA transfers.

```
char *addr;

addr = get1page ();
if (addr == NULL) {
    pseterr (ENOMEM);
    return (SYSERR);
}
free1page (addr);
```

alloc_cmem/free_cmem function similarly to get1page/free1page but allocate a number of physically contiguous pages. This is useful for DMA devices that do not support scatter/gather techniques. Although alloc_cmem takes as argument a number of bytes, this is rounded up to an integral number of pages and the returned address is aligned on a page boundary.

```
int nbytes;
char *addr;

addr = alloc_cmem (nbytes);
if (addr == NULL) {
    pseterr (ENOMEM);
    return (SYSERR);
}
free_cmem (addr, nbytes);
```

Both alloc_cmem and get1page return NULL if no memory is available. It is the driver’s responsibility to ensure that the arguments passed to free_cmem/free1page correspond to memory previously allocated with alloc_cmem/get1page.

DMA Transfers

The fact that LynxOS uses the CPU's MMU makes the programming of DMA transfers slightly more complicated. All addresses generated by the CPU are treated as virtual and are converted to physical addresses by the MMU. Memory that is contiguous in virtual space can be mapped to non-contiguous physical pages. DMA devices, however, typically work with physical addresses. Therefore, a driver must convert virtual addresses to their corresponding physical addresses before passing them to a DMA controller.

In addition, user memory may be paged, which can lead to changes in the virtual to physical address mapping and to a physical page being reallocated to another user task. Paging must therefore be suppressed on user memory that is involved in a DMA transfer by locking the memory region. When the DMA transfer is complete, the memory should be unlocked.

Memory Locking

LynxOS provides a pair of routines for locking and unlocking user memory, `mem_lock`/`mem_unlock` and `dma_lock`/`dma_unlock`. If paging has not been activated, either by the `vmstart` command or system call, these routines have no effect.

Locking Memory

```
/* Lock user memory to prevent paging */

if (mem_lock (pid, uaddr, size) == SYSERR) {
    pseterr (ENOMEM);
    return (SYSERR);
}
```

The arguments *uaddr* and *size* specify the address and size of a memory region in the process specified by *pid*. These will normally correspond to the arguments passed to the driver read or write entry point by the user task. A driver can call the kernel function `getpid()` to get the process ID of the current task.

If paging is activated and certain virtual addresses in the specified range are not currently mapped to real memory (i.e. they have been paged out), `mem_lock` and `dma_lock` will attempt to allocate and map in physical memory. If there is not enough physical memory available to do this, the error code SYSERR is returned. Otherwise, the routines return OK, which can be taken to mean that the specified virtual address range is now mapped to physical memory and is locked.

It is permissible to lock the same address multiple times. This may occur, for example, when locking two different but overlapping regions. However, the memory must be unlocked the same number of times.

Note that kernel memory is not paged. Therefore, memory allocated with `sysbrk`, `get1page` or `alloc_cmem` does not require locking.

Unlocking Memory

```
/* Unlock user memory */

if (mem_unlock (pid, uaddr, size, dirty) == SYSERR) {
    return (SYSERR);   /* error set to EINVAL by
                                mem_unlock */
}
```

If the memory contents have been modified by another processor, such as a DMA controller, the *dirty* flag should be set to 1. This will ensure that the contents are written out to the swap file if the page is paged out.

Once memory is unlocked it becomes eligible for paging again. Attempting to unlock memory that was not previously locked will cause the routine to return `SYSERR`. A driver should always ensure that any locked memory is released when a task closes the device or when the device is uninstalled. Pages that remain locked after a task has exited, though usable by other tasks, will not be paged.

Address Translation

A virtual address can be converted to its physical addresses using `get_phys`.

```
#include <kernel.h>

physaddr = (get_phys (virtaddr) - PHYSBASE + drambase);
```

The address returned is, in fact, a kernel direct mapped addresses. To convert this address to its physical address, the constant PHYSBASE must be subtracted and `drambase` added. The variable `drambase` contains the physical address of the start of RAM. On most platforms, this will be 0, but for maximum portability, it should be used in the calculation. A driver should never modify the value of `drambase`!

The returned address is only valid up to the next page boundary as contiguous virtual addresses do not necessarily map to contiguous physical addresses. To convert virtual address ranges that cross page boundaries, *mmchain* should be used.

```
#include <mem.h>
#include <kernel.h>

struct dmachain array[NCHAIN];

mmchain (array, virtaddr, nbytes);
for (i = 0; i < nsegments; i++) {
    physaddr = array[i].address - PHYSBASE + drambase;
    length = array[i].count;
    ...
}
```

The virtual memory region is defined by its address (virtaddr) and size (nbytes). *mmchain* fills in the fields in the dmachain array with the physical addresses and sizes of the physical memory segments which make up the specified virtual address range. The first and last segments in the list may be less than a whole page in size, as illustrated in Figure 3-1.

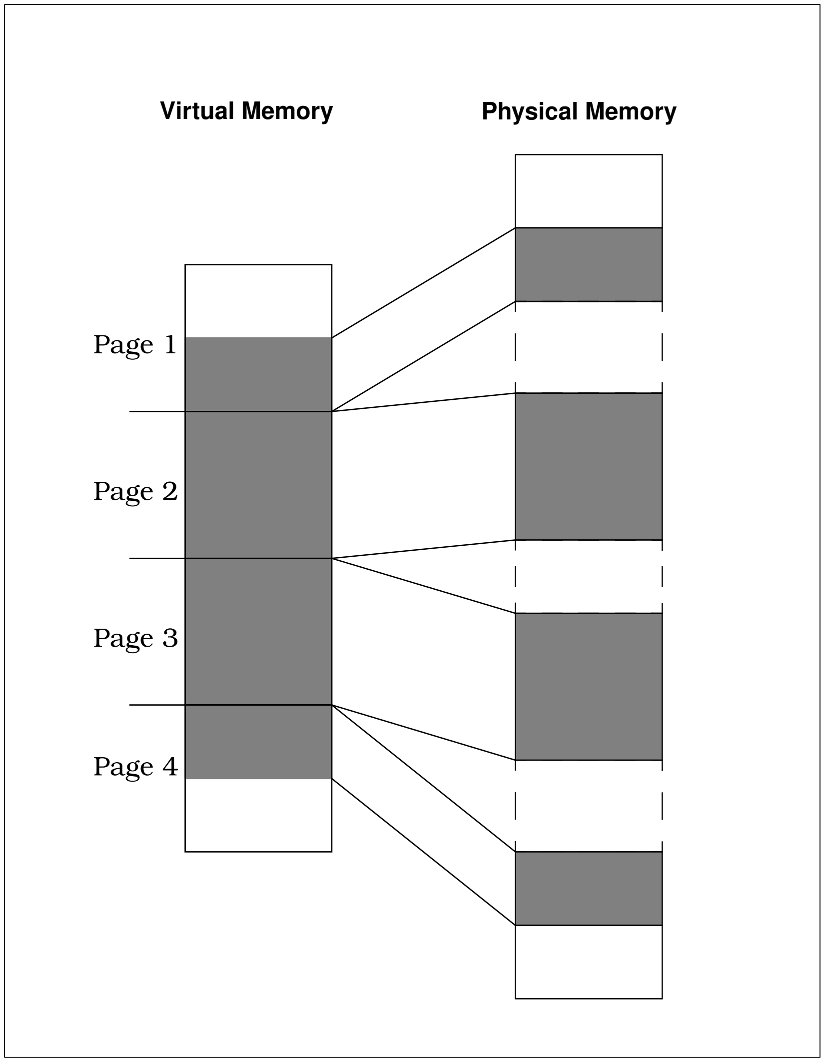

Figure 3-1: Mapping Virtual to Physical Memory

It is the responsibility of the driver to ensure that the `dmachain` array is large enough to hold all the segments that make up the specified address range. The maximum number of segments can be calculated as follows:

```
nsegments = (nbytes + PAGESIZE - 1) / PAGESIZE + 1;
```

`mmchain` returns the actual number of segments.

Both `get_phys` and `mmchain` translate addresses in the context of the current process. In the case where no physical memory is mapped to a virtual address, both `get_phys` and `mmchain` set the converted address to 0. `mem_lock` is therefore normally used before the address conversion routine to ensure that there is a valid mapping. To translate addresses of a different process, `mmchainjob` can be used. This takes an additional argument, *job*, which identifies the *pid* of the target process.

```
mmchainjob (job, array, virtaddr, nbytes);
```

Accessing User Space from Interrupt Handlers and Kernel Threads

Interrupt handlers and kernel threads execute asynchronously with respect to the user task making requests to the driver. An interrupt handler executes in the context of the task that was current when the interrupt occurred. Kernel threads execute in the context of the null process (process 0). Since the null process has no user context associated with it, the switch to a kernel thread is much quicker than to a user thread. It is sometimes necessary for an interrupt handler or a kernel thread to access a location in a user task, even though the target task may not be the currently mapped task. This requires some special considerations. The key to understanding how this can be done is the fact that there is a second virtual address that can be used to access an address in user space.

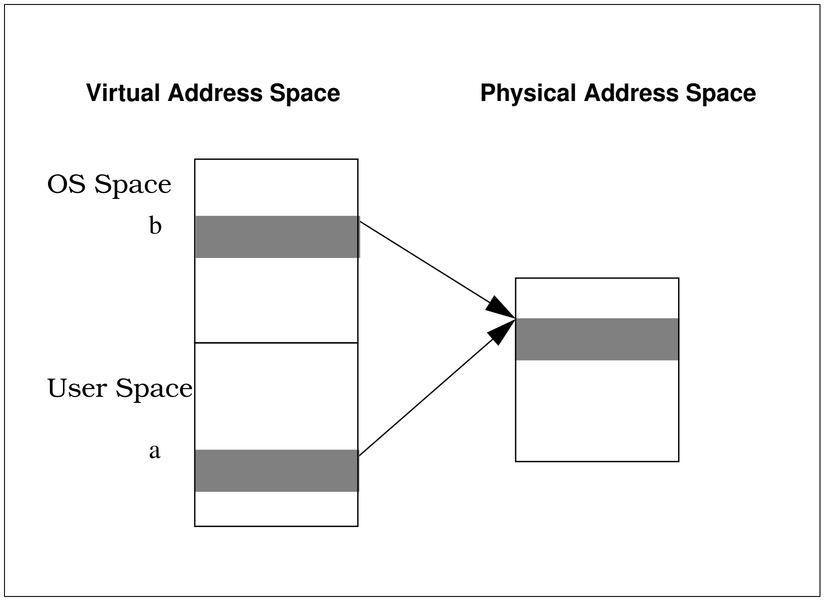

Figure 3-2: Aliasing a User Virtual Address

As Figure 3-2 shows, for any address in user space there are in fact two virtual addresses mapped to the physical address. One is the user virtual address (a). However, this mapping is only valid when the user task is the current task. So it can not be used from an interrupt handler or kernel thread. This mapping may also be changed if paging is activated. The second mapping is the direct mapped kernel virtual address (b). This mapping is permanent so it can safely be used anywhere in a driver.

Care must therefore be taken when accessing user space from an interrupt handler or kernel thread. The driver must first convert the user virtual address to its corresponding kernel direct mapped address in the top-half routine (usually the ioctl entry point) and then pass this address to the interrupt handler or kernel thread via the statics structure. The user address must also be locked to prevent the address mapping from being changed by the paging system.

In the following example, *u_addr* and *size* specify the virtual address and size of the user memory to be accessed and would be passed to the driver from the application.

```
drvioctl (s, f, cmd, arg)
struct statics *s;
struct file *f;
char *arg;
{
    char *kvaddr;

    . . . .
    if (mem_lock (getpid (), u_addr, size) == SYSERR) {
        pseterr (ENOMEM);
        return (SYSERR);
    }
    kvaddr = get_phys (u_addr);    /* convert user addr */
    s->u_status = kvaddr;          /* save for later use */
    s->u_size = size;
    s->u_addr = u_addr;    /* used for unlocking memory */
    s->pid = getpid ();
    . . . .
}
```

The pointer can now used from an interrupt handler or kernel thread to access the user memory.

```
kernel_thread (s)
struct statics *s;
{
    ...
    if (s->u_status) {
       *(s->u_status) = status;    /* pass info to user
                                             task */
        /* unlock user memory */
        mem_unlock (s->pid, s->u_addr, s->size, 0);
        s->u_status = NULL);
    }
    ...
}
```

CHAPTER 4

Synchronization

Introduction

Synchronization is probably the most important aspect of a device driver. It is also the most intricate and therefore most difficult to master. It requires a lot of careful thought and attention and is more often than not the cause of the most frustrating bugs that take hours or days to locate but only seconds to fix. This aspect of a driver can make or break not only the driver but the whole system. Poorly designed synchronization, while it may not crash the system, can nevertheless have a disastrous effect on the system's real-time performance.

For those new to writing device drivers or those not familiar with pre-emptible kernels, here is a review of the problems that give rise to the need for synchronization. This discussion is not specific to LynxOS, so those already familiar with the subject may skip to “Synchronization Mechanisms” on page 41.

Event Synchronization

The most frequent type of synchronization found in device drivers is the need to wait for a device to complete a data transfer or other operation. Another typical event is waiting for a resource, such as a buffer, to become free. The simplest technique is to use polling but this is unsuitable in a real-time system (except in some very specific situations) for a number of reasons:

- CPU time is used unnecessarily
- response time is undeterministic because the event occurs asynchronously relative to the polling loop
- if a higher priority task is running, the driver can not poll, so data may be lost

The use of interrupts overcomes these drawbacks, but a driver needs a way to wait for an interrupt to occur without using CPU cycles. Kernel semaphores are the mechanism used to allow a driver to block and to be woken up when the event occurs, as illustrated in the Figure 4-1.

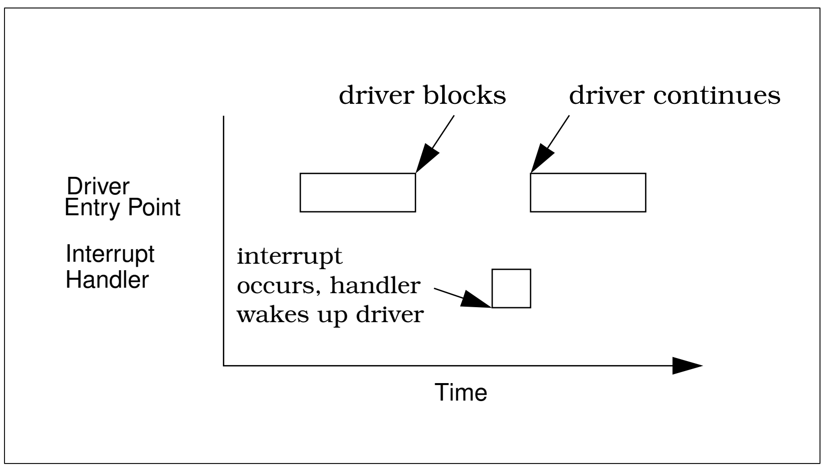

Figure 4-1: Event Synchronization

Critical Code Regions

The LynxOS kernel, being designed for real-time, is completely pre-emptible, including the device drivers. This means that one task can pre-empt another, even if the pre-empted task happens to be in the middle of a system call such as a driver entry point.

Consider the following code, which is an attempt to provide exclusive access to a device:

```
1   if (device_free == TRUE) {
2       device_free = FALSE;
3       ...                  /* use device */
4       device_free = TRUE;
5   }
```

Imagine a system where two tasks are using the device. The first task executes line 1 of the above code and finds the device to be free. But before being able to execute line 2, it is pre-empted by task 2 which then executes the same code. The second task also finds the device free (the first task hasn't executed line 2 yet) and so proceeds to use it. While task 2 is using the device it is pre-empted by task 1 (maybe because its time quantum ran out). Task 1 continues where it left off and proceeds to access the device as well. So both tasks are accessing the device concurrently, just the situation that the code was supposed to prevent!

This type of situation is referred to as a race condition, i.e. the result of the computation depends on the timing of task execution. A race condition is normally considered a bug! Concurrent accesses to a device may well lead to a system crash. The example also serves to illustrate the insidious nature of this type of problem. The probability that a pre-emption occurs between lines 1 and 2 is obviously quite low so the problem will occur very infrequently and can be extremely difficult to track down or reproduce systematically.

Code that accesses shared resources, such as a device, is known as a *critical code region* and the mechanism used to synchronize access to the code is often referred to as a *mutex* as it provides mutual exclusion.

In order to avoid race conditions, drivers must be made re-entrant. This means that critical code regions must be protected such that access to shared resources is serialized i.e. only one task at a time can be executing the code. There are a number of mechanisms for protecting critical code regions which will be discussed in the following sections.

When considering synchronization, the interrupt handler must also be taken into account. It too can pre-empt the current task at any moment and may access static data within the driver. But because an interrupt handler is not scheduled in the same way as other tasks, the mechanisms used to synchronize with it are usually different from those used to synchronize between user tasks. Even a non pre-emptible kernel such as Unix, where synchronization between tasks is not an issue, must nevertheless provide a mechanism for synchronizing with interrupt handlers.

Figure 4-2 summarizes the problems of synchronizing critical code regions in a multi-tasking kernel.

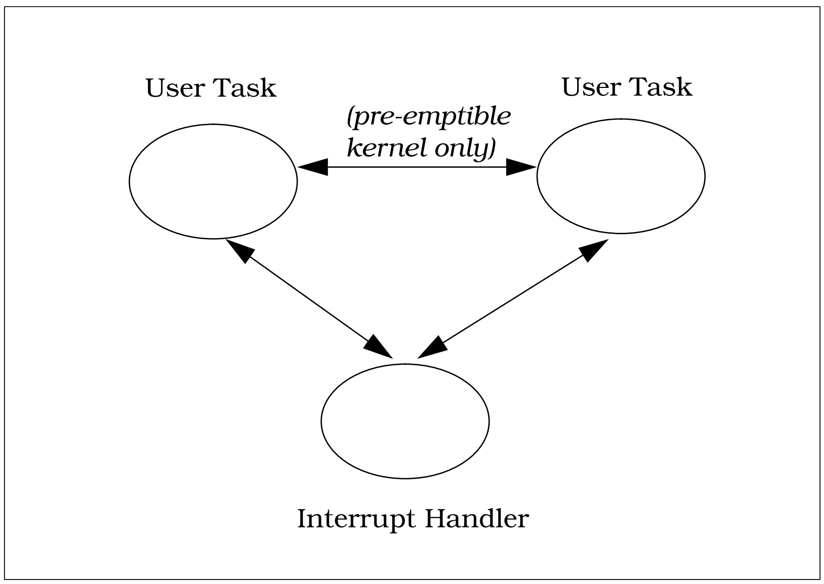

Figure 4-2: Problems in Synchronizing Critical Code Regions in a Multi-Tasking Kernel

Resource Pool Management

A third type of synchronization involves the dynamic management of a shared pool of resources, such as a set of data buffers. A counting semaphore is used to atomically remove items from and return items to the central pool. The remove operation may block if no resources are currently available and the return operation wakes up any tasks waiting for resources.

Synchronization Mechanisms

There are a number of synchronization mechanisms that can be used in a LynxOS device driver:

- kernel semaphores
- disabling pre-emption
- disabling interrupts

Broadly speaking, we can place the mechanisms on a scale of *selectivity*, which refers to the level of impact the use of the mechanism has on other activity in the system. Kernel semaphores are the most selective and only affect tasks that are using the driver. There is no impact on other non-related tasks or activities in the system. Disabling pre-emption is obviously less selective as it will have an impact on all tasks running in the system, regardless of whether they are using the driver. Higher priority tasks will be prevented from executing until pre-emption is re-enabled. The system will continue to handle interrupts however. Disabling interrupts is the least selective method. Both task pre-emption and interrupt handling are disabled so the only activity that can occur in the system is the execution of the driver that disabled interrupts. This technique must be used with care!

Which mechanism is used depends on a number of factors. The different synchronization mechanisms and the situations in which each should be used will be discussed in detail below. First, let's describe the mechanisms themselves.

Kernel Semaphores

General Characteristics

These are the most versatile and widely used driver synchronization mechanism. A kernel semaphore is simply an integer variable that is declared by the driver. All semaphores should be part of the statics structure, allocated and initialized in the install routine. Never use a local stack variable as a kernel semaphore.

Kernel semaphores are counting semaphores, they can be initialized to any non-negative value. A semaphore is acquired using the `swait` function.

```
swait (&sem, flag);
```

Note that the address of the semaphore is passed to `swait`. The *flag* argument is explained below.

If the semaphore value is greater than zero, it is simply decremented and the task continues. If the semaphore value is less than or equal to zero, the task blocks and is put on the semaphore's wait queue. Tasks on this queue are kept in priority order.

A semaphore is signaled using the `ssignal` function, which like `swait`, takes the semaphore address as argument.

```
ssignal (&sem);
```

If there are tasks waiting on the semaphore's queue, the highest priority task is woken up. Otherwise the semaphore value is incremented.

Kernel semaphores have state. The semaphore's value “remembers” how many times the semaphore has been waited on or signaled. This is important for event synchronization. If an event occurs but there are no tasks waiting for that event, the fact that the event occurred is not forgotten.

Kernel semaphores are not “owned” by a particular task. Any task can signal a semaphore, not just the task that acquired it. This is necessary to allow kernel semaphores to be used as an event synchronization mechanism but requires care when the semaphore is used for mutual exclusion.

The *flag* argument to the `swait` function allows a task to specify how signals are handled while it is blocked on a semaphore. If the task does not block, this argument is not used. There are three possibilities for *flag*, specified using symbolic constants defined in `<kernel.h>`:

`SEM_SIGIGNORE`	Signals have no effect on the blocked task. Any signals sent to the task while it is waiting on the semaphore remain pending and will be delivered at some future time.

`SEM_SIGRETRY`	Signals are delivered to the task. If the task's signal handler returns, the task automatically waits again on the semaphore. Signal delivery is transparent to the driver as the `swait` function does not indicate whether any signals were delivered.
`SEM_SIGABORT`	If a signal is sent to the task while it is blocked on the semaphore, the `swait` is aborted. The task is woken up and `swait` returns a non-zero value. The signal remains pending.

Other Kernel Semaphore Functions

There are a number of other functions used to manipulate kernel semaphores.

`ssignal`*n*	Used to signal a semaphore *n* times. This is equivalent to calling `ssignal` *n* times.
`sreset`	Resets the semaphore value to 0 and wakes up all tasks that are waiting on the semaphore.
`scount`	Returns the semaphore value. A negative count indicates the number of tasks that are waiting for the semaphore. This function is used rarely in a driver but if the need arises, it should always be used rather than using the value of the semaphore variable directly. The latter technique would give erroneous results because the value is not a simple count when there are tasks blocked on the semaphore.

The use of these routines is further illustrated in the examples discussed below.

Disabling Interrupts

Lynx provides routines for disabling/enabling all interrupts. Similarly, during the execution of an interrupt handler, the kernel disables all interrupts. A frequently asked question is why this is so, rather than allowing masking of certain interrupt levels. There are several reasons:

- Portability. LynxOS is a portable OS running on many different CPU architectures. Device Drivers also need to be portable. Masking of certain interrupt levels is only available on some CPU architectures. Disabling all interrupts is probably the only portable technique.
- Disabling certain interrupt levels may require accessing an off-chip interrupt controller which can be a time-consuming operation, maybe taking more time than the actual code that needs to be protected. Many RISC architectures provide a single instruction for masking all interrupts which executes very rapidly.
- The major need for masking specific interrupt levels was to implement interrupt “nesting”, allowing higher priority interrupts to interrupt a lower priority interrupt. This need arose because of the existence of long interrupt handlers. Without nesting, a long interrupt routine for a low priority interrupt could block out a higher priority interrupt. LynxOS kernel threads overcome this problem. The interrupt service routines are always very short so disabling all interrupts during that time does not have undesirable consequences.

The routines for controlling interrupts are `disable` and `restore` and are used as follows:

```
int ps;


disable (ps);     /* disable all interrupts */
    ...
    ...
restore (ps);     /* restore interrupt state */
```

The variable *ps* must be a local variable and should never be modified by the driver. Each call to `disable` must have a corresponding call to `restore`, using the same variable. Note that `restore` does not necessarily re-enable interrupts. Rather, it restores the state that held before `disable` was called. So if interrupts were already disabled when `disable` was called, the (first) call to `restore` will not re-enable them. As a consequence of disabling interrupts, task pre-emption is also disabled.

Disabling Pre-Emption

The kernel routines `sdisable`/`srestore` control task pre-emption and are used in much the same way as `disable`/`restore`.

```
int ps;

sdisable (ps);      /* disable task pre-emption */
    ...
    ...
srestore (ps);      /* restore pre-emption state */
```

The variable *ps* must be a local variable and should never be modified by the driver. Each call to `sdisable` must have a corresponding call to `srestore`, using the same variable. Note that `srestore` does not necessarily re-enable pre-emption. Rather, it restores the state that held before `sdisable` was called. So if pre-emption was already disabled when `sdisable` was called, the (first) call to `srestore` will not re-enable it. The kernel continues to handle interrupts while pre-emption is disabled.

Critical Code Regions

If the shared resource is accessed by the interrupt handler then `disable`/`restore` must be used to protect critical code regions. If not, then either of the three synchronization mechanisms can be used, the kernel semaphore being preferred because of its selectivity.

Using Kernel Semaphores for Mutual Exclusion

When used to protect a critical code region, the kernel semaphore should be initialized to 1. This allows the first task to lock the semaphore and enter the region. Other tasks (including a kernel thread) that attempt to enter the same region will block until the semaphore is unlocked. Each call to `swait` must have a corresponding call to `ssignal`.

```
swait (&s->mutex, SEM_SIGIGNORE); /* enter critical
                                      code region */
    ...
    ...                            /* access resource */
    ...
ssignal (&s->mutex);               /* leave critical
                                      code region */
```

Signals can normally be ignored when using a kernel semaphore as a mutex. Compared to waiting for an I/O device, a critical code region is relatively short so there is little need to be able to interrupt a task that is waiting on the mutex. Unlike an event, which is never guaranteed to occur, execution of a critical code region can not “fail”. The task holding the mutex is bound, sooner or later, to get to the point where the mutex is released.

CAUTION! `sreset` *and* `ssignaln` *should never be used on a kernel semaphore that is used for mutual exclusion as in both cases this could lead to more than one task executing the critical code concurrently.*

Priority Inheritance Semaphores

In a multi-tasking system that uses a fixed priority scheduler, a problem known as *priority inversion* can occur. Consider a situation where a task holds some resource. This task is pre-empted by a higher priority task which requires access to the same resource. The higher priority task must wait until the lower priority task releases the resource. But the lower priority task may be prevented from executing (and therefore from releasing the resource) by other tasks of intermediate priority. One solution to this problem is to use *priority inheritance* whereby the priority of the task holding the resource is temporarily raised to the priority of the highest priority task waiting for that resource until it releases the resource. LynxOS kernel semaphores support priority inheritance. In order to function with priority inheritance, the semaphore's value must be initialized to -1. Alternatively, the kernel function `pi_init` can be used:

```
pi_init (&s->mutex);
```

The use of this feature is only meaningful in the context of a kernel semaphore being used as a mutex mechanism.

Using Interrupt Disabling for Mutual Exclusion

`disable`/`restore` must be used for synchronization with an interrupt handler. Note that within the interrupt handler itself, interrupts are already disabled, so calls to `disable`/`restore` are not required. One resource that is almost always accessed by the interrupt handler are the device registers. All code that accesses these should be protected using `disable`/`restore`. In addition, there may be some timing constraints on the way in which the registers are accessed which is another good reason to use `disable`/`restore`.

`disable`/`restore` can also be used for mutual exclusion between user tasks (and kernel threads) but, as always, they should be used with caution. Their advantage is that they incur less overhead than `swait`/`signal`. It should be remembered that `swait` only incurs significant overhead if the semaphore is locked. If the semaphore is free, the execution time for `swait` is very fast. If the semaphore is already locked, `swait` will put the requesting task into the blocked state and then call the scheduler to perform a context switch. Similarly, `ssignal` only incurs significant overhead if there are tasks blocked on the semaphore. Using `disable`/`restore`, the overhead of blocking the task and the context switch is saved. They should be used when the critical code region is very short. A typical example is taking an item off (or putting an item on) a free list:

```
disable (ps);
if (ptr = freelist)      /* take item off free list */
    freelist = ptr->next;
restore (ps);

disable (ps);
ptr->next = freelist;   /* put item back on free list */
freelist = ptr;
restore (ps);
```

If a driver allows multiple tasks to use a device concurrently and the critical region is in part of the driver that will be executed very often, then there may be some advantage in using `disable/restore` in order to avoid an excessive number of context switches. But if the driver is written so that only one user task can use the device at a time, then `swait/ssignal` will probably be sufficient for synchronization with the driver's kernel thread.

When using `disable/restore`, it is essential that the time during which interrupts are disabled is kept to a minimum to avoid having an impact on the system's response times. It is sometimes possible to split a seemingly long critical code section into two (or more pieces) by introducing calls to `restore/disable` at appropriate points. Care must be taken in selecting the point at which interrupts are re-enabled to avoid creating a race condition.

```
disable (ps);
    ...
    ...
restore (ps);    /* allow interrupts and pre-emption */
disable (ps);
    ...
    ...
restore (ps);
```

If the critical code region is still too long then the driver should be redesigned so that `swait/ssignal` can be used and another means found for the interrupt handler to communicate with the rest of the driver.

Another reason to use `disable`/`restore` is to avoid deadlock situations. These situations are discussed in “Combining Synchronization Mechanisms” on page 55.

Using Pre-emption Disabling for Mutual Exclusion

If the shared resource is not accessed by the interrupt handler, `sdisable`/`srestore` can be used instead of `disable`/`restore`. This allows the system to continue to handle interrupts during the critical code region. The same remarks and considerations that apply to the use of `disable`/`restore` apply equally to `sdisable`/`srestore`.

Event Synchronization

A kernel semaphore is the mechanism used to implement event synchronization in a LynxOS driver. The value of the semaphore should be initialized to 0, indicating that no events have occurred.

Waiting for an event:

```
if (swait (&s->event_sem, SEM_SIGABORT)) {
    pseterr (EINTR);
    return (SYSERR);
}
```

Signaling an event:

```
ssignal (&s->event_sem);
```

Handling Signals

Because there is often no guarantee that an event will occur, signals should be allowed to abort the swait using `SEM_SIGABORT`. This way, a task can be interrupted if the event it is waiting for never arrives. If signals are ignored, there will be no way to interrupt the task in the case of problems, so the task could remain in the blocked state indefinitely. The driver must check the return code from `swait` to determine whether a signal was received. As an alternative to

SEM_SIGABORT, timeouts can be used if the timing of events is known in advance.

It is sometimes useful for an application to be able to handle signals while it is blocked on a semaphore but without aborting the wait. This is possible using the SEM_SIGRETRY flag to swait. Signals are delivered to the application and the swait automatically restarted. There is no way for the driver to know whether any signals were delivered while the task was blocked on the semaphore.

A word of caution is necessary concerning the use of SEM_SIGRETRY. If the signal handler in the application calls exit(3), then the swait in the driver will never return. This could cause problems if the task had blocked while holding some resources. These resources will never be freed. To avoid this type of problem, a driver can use SEM_SIGIGNORE in conjunction with the kernel function deliversigs. This allows the application to receive signals in a timely fashion, but without the risk of losing resources in the driver.

```
if (swait (&s->event_sem, SEM_SIGABORT) {
   cleanup (s); /* tidy up before delivering signals */
    deliversigs (); /* may never return */
}
```

Using sreset with Event Synchronization Semaphores

Two example uses of sreset discussed below are:

1. In handling error conditions.
2. Variable length data transfers (with multiple consumers).

Handling Error Conditions

A driver must handle errors that may occur. For example, what should it do if an unrecoverable error is detected on a device? A frequent approach is to set an error flag and wake up any tasks that are waiting on the device:

```
if (error_found) {
    s->error++;
    sreset (&s->event_sem);
}
```

But now the driver can not assume that when `swait` returns, the expected event has occurred. The `swait` could have been woken up because an error was detected. So some extra logic is required when using the event synchronization semaphore:

```
if (swait (&s->event_sem, SEM_SIGABORT)) {
    pseterr (EINTR);
    return (SYSERR);
}
if (s->error) {
    pseterr (EIO);
    return (SYSERR);
}
```

Variable Length Transfers

The second example using `sreset` is somewhat more esoteric but is an interesting example nevertheless.

Imagine the following scenario. A device or “producer” process generates data at a variable rate. Data can also be consumed in variable sized pieces by multiple tasks. At some point, a number of consumer tasks may be blocked on an event synchronization semaphore, each waiting for different amounts of data, as illustrated in Figure 4-3.

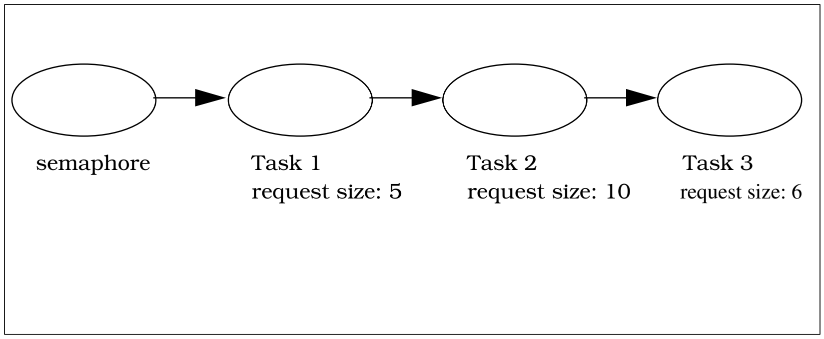

Figure 4-3: Synchronization Mechanisms

When some data become available, what should the driver do? Without adding extra complexity and overhead to the driver, there is no easy way for the driver to calculate how many of the waiting tasks it can satisfy (and should therefore wake up). A simple solution is to call `sreset` which will wake all tasks which will then consume the available data according to their priorities. Tasks that are awaken but find no data left will have to wait again on the event semaphore.

Caution When Using sreset

To maintain coherency of the semaphore queue, `sreset` must synchronize with calls to `ssignal`. Because `ssignal` can be called from an interrupt handler, `sreset` disables interrupts internally while it is waking up all the blocked tasks. Since the number of tasks blocked on a semaphore is not limited this could lead to unbounded interrupt disable times if `sreset` is used without proper consideration.

To avoid this problem, another technique must be used in driver designs where an unknown number of tasks could be blocked on a semaphore. One possibility is to wake tasks in a “cascade”. The call to `sreset` is replaced by a call to `ssignal`, which will wake up the first blocked task. This task is then responsible for unblocking the next blocked task, which will unblock the next one etc., until there are no more blocked tasks. A negative semaphore indicates that there are blocked tasks. This is illustrated in the modified error handling code from the previous section:

```
if (error_found) {
    s->error++;
    if (s->event_sem < 0)
        ssignal (&s->event_sem);
}

    ...

if (swait (&s->event_sem, SEM_SIGABORT)) {
    pseterr (EINTR);
    return (SYSERR);
}
if (s->error) {
    if (s->event_sem < 0)
        ssignal (&s->event_sem);
    pseterr (EIO);
    return (SYSERR);
}
```

Note that because tasks are queued on a semaphore in priority order, they will still be awaken and executed in the same order as when using `sreset`. There is no penalty to using this technique.

Resource Pool Management

LynxOS kernel semaphores can also be used as a counting semaphore for managing a resource pool. The value of the semaphore should be initialized to the number of resources in the pool. To allocate a resource, `swait` is used. `ssignal` is used to free a resource.

The following code fragment shows an example of using `swait` to allocate and `ssignal` to free a resource.

```
struct resource *
allocate (s)
struct statics *s;
{
    struct resource *resource;
    int ps;

    swait (&s->pool_sem, SEM_SIGRETRY);
    sdisable (ps);
    resource = s->pool_freelist;
    s->pool_freelist = resource->next;
    srestore (ps);
    return (resource);
}

free (s, resource)
struct statics *s;
struct resource *resource;
{
    struct resource *resource;
    int ps;

    sdisable (ps);
    resource->next = s->pool_freelist;
    s->pool_freelist = resource;
    srestore (ps);
    ssignal (&s->pool_sem);
}
```

The counting semaphore functions implicitly as an event synchronization semaphore too. When the pool is empty an attempt to allocate will block until another task frees a resource.

Note that a mutex mechanism is still needed to protect the code that manipulates the free list. The combining of different synchronization techniques is discussed more fully in the following section.

Combining Synchronization Mechanisms

The examples discussed in the preceding sections have all been fairly straightforward in that they have only used one synchronization mechanism. In a real driver the scenarios are often far more complex and require the combining of different techniques. The following sections discuss when and how the synchronization mechanisms should be combined.

Manipulating a Free List

Refer to the code in “Using Interrupt Disabling for Mutual Exclusion” on page 47. This illustrates the use of interrupt disabling to remove an item from a free list, but does not address what the driver can do if the free list is empty. One possibility is that the driver blocks until another task puts something back on the free list. This scenario requires the use of a mutex and an event synchronization semaphore.

Two different approaches to this problem are illustrated in the following examples:

```
/* get_item : get item off free list, blocking if
list is empty  */
struct item *
get_item (s)
struct statics *s;
{
    struct item *p;
    int ps;

    do {
        disable (ps);        /* enter critical code */
       if (p = s->freelist) /* take 1st item on list */
            s->freelist = p->next;
        else
            /* list was empty, so wait */
          swait (&s->freelist_sem, SEM_SIGIGNORE);
        restore (ps);        /* exit critical code */
    } while (!p);
    return (p);
}

/* put_item : put item on free list, wake up waiting
   tasks  */
put_item (s, p)
struct statics *s;
struct item *p;
{
    int ps;

    disable (ps);            /* enter critical code */
    p->next = s->freelist;    /* put item on list */
    s->freelist = p;
    if (s->freelist_sem < 0)
      ssignal (&s->freelist_sem);  /* wake up waiter */
    restore (ps);             /* exit critical code */
}
```

There are a number of points of interest illustrated by this example:

1. The example uses `SEM_SIGIGNORE` for simplicity. If `SEM_SIGABORT` is used, the return value from `swait` must be checked.

2. The example uses the `disable/restore` mechanism for mutual exclusion. This allows the free list to be accessed from an interrupt handler using `put_item`. `get_item` should never be called from an interrupt handler though, as it may block. If the free list is not accessed by the interrupt handler, `sdisable/srestore` can be used instead.

3. The `get_item` function uses the value of the item taken off the list to determine if the list was empty or not. Note that the `freelist_sem` is being used simply as an event synchronization mechanism, not a counting semaphore. (Managing a free list with a counting semaphore is illustrated in the second approach). As a consequence, the code that puts items back on the free list must *only* signal the semaphore if there is a task waiting. Otherwise, if the semaphore was signalled every time an item is put back, the semaphore count would become positive and a task calling `swait` in `get_item` would return immediately, even though the list is still empty.

4. Blocking with interrupts disabled may seem at first sight like a dangerous thing to do. But this is necessary as restoring interrupts before the `swait` would introduce a race condition. LynxOS saves the interrupt state on a *per task* basis. So when this task blocks and the scheduler switches to another task, the interrupt state will be set to that associated with the new task. But from the point of view of the task executing the above code, the `swait` executes atomically with interrupts disabled.

5. `swait/ssignal` can't be used as the mutex mechanism in this particular example as this could lead to a deadlock situation where one task is

blocked in the `swait` while holding the mutex. Other tasks wishing to put items back on the list will not be able to enter the critical region. If a critical code region may block care must be taken not to introduce the possibility of a deadlock. To avoid a deadlock, `sdisable/srestore` or `disable/restore` should be used as the mutex mechanism rather than `swait/ssignal`, but once again, the critical code region must be kept as short as possible to avoid having any adverse effect on the system's real-time responsiveness. An alternative would be to raise an error condition if the list is empty rather than block. This would allow `swait/ssignal` to be used as the mutex mechanism.

6. A call to `ssignal` in `put_item` may make a higher priority task eligible to execute but the context switch won't occur until pre-emption is re-enabled with `restore`.

In the second approach to this problem, a kernel semaphore is used as a counting semaphore to manage items on the free list. The value of the semaphore should be initialized to the number of items on the list.

```
struct item *
get_item (s)
struct statics *s;
{
    struct item *p;
    int ps;

    swait (&s->free_count, SEM_SIGRETRY);
    disable (ps);
    p = s->freelist;
    s->freelist = p->next;
    restore (ps);
    return (p);
}

put_item (s, p)
struct statics *s;
struct item *p;
{
    int ps;

    disable (ps);
    p->next = s->freelist;
    s->freelist = p;
    restore (ps);
    ssignal (&s->free_count);
}
```

This code illustrates a number of points:

1. A kernel semaphore used as a counting semaphore incorporates the functionality of an event synchronization semaphore. `swait` will block when no items are available and `ssignal` will wake up waiting tasks.
2. The example uses the `disable`/`restore` mechanism for mutual exclusion. This allows the free list to be accessed from an interrupt handler using `put_item`. `get_item` should never be called from an interrupt handler though, as it may block. If the free list is not accessed by the interrupt handler, `sdisable`/`srestore` can be used instead.

3. The code logic is simpler in this second example. However, there is the extra overhead of calling `swait` or `ssignal` for every item.
4. The event synchronization is outside of the critical code region so there is no possibility of deadlock. Therefore, `swait`/`ssignal` could be used as the mutex mechanism if the code does not need to be called from an interrupt handler.
5. The function `put_item` could be modified to allow several items to put back on the list using `ssignal`*n*. But items can only be consumed one at time, there is no function `swait`*n*.

Signal Handling and Real-Time Response

"Handling Signals" on page 49 discussed the use of the `SEM_SIGRETRY` flag with `swait`. It is not advisable to use `swait` with this flag inside a critical code region protected with `(s)disable`/`(s)restore`. The reason for this is that, internally, `swait` calls the kernel function `deliversigs` to deliver signals when the `SEM_SIGRETRY` flag is used. If the `swait` is within a region with interrupts or pre-emption disabled, then the execution time for `deliversigs` will contribute to the total interrupt or pre-emption disable time, as illustrated in the following example:

```
sdisable (ps);      /* enter critical region */
    ...
swait (&s->event_sem, SEM_SIGRETRY);
                 /* may call deliversigs internally */
    ...
srestore (ps);      /* leave critical region */
```

In order to minimize the disable times it is better to use SEM_SIGABORT and re-enable interrupts/pre-emption before calling deliversigs. The above code then becomes:

```
sdisable (ps);             /* enter critical region */
    ...
while (swait (&s->event_sem, SEM_SIGABORT)) {
    srestore (ps);       /* re-enable pre-emption
                           before delivering signals */
    deliversigs ();        /* may never return */
    sdisable (ps);
}
    ...
srestore (ps);             /* leave critical region */
```

Nesting Critical Regions

It is also possible to nest critical regions. As a general rule, a less selective mechanism can be nested inside a more selective one. For instance, the following is permissible:

```
int sps, ps;

sdisable (sps);
     ...
disable (ps);
     ...
restore (ps);
     ...
srestore (sps);
```

Note that different local variables must be used for the two mechanisms. However, the converse is not true. It is not permitted to do the following:

```
disable (ps);
     ...
sdisable (sps);
     ...
srestore (sps);
     ...
restore (ps);
```

In any case, the inner sdisable/srestore is completely redundant, as pre-emption is already disabled by the outer disable.

CHAPTER 5

Accessing Hardware

Device drivers provide the ability to synchronize access and control of the operating system to the hardware. This chapter describes how device drivers access the hardware layer and illustrates the virtual memory mappings used by LynxOS on different hardware platforms.

General Tasks

Synchronization

Lynx recommends that you protect the access to device registers by disabling interrupts. For more information, see Chapter 7, "Interrupt Handling".

Handling Bus Errors

An access to a memory location where no device is responding may cause the hardware to generate a bus error. By default, LynxOS will halt, printing some diagnostic information on the debug port, when a bus error occurs. To change this behavior, a driver can catch bus errors using the `recoset`/`noreco` routines. These routines should surround the code that could potentially cause a bus error.

```
if (recoset ()) {
    pseterr (EFAULT);    /* a bus error occured */
   noreco();          /* must restore before leaving */
    return (SYSERR);
}
...                        /* access device */
noreco ();               /* restore default behavior */
```

When first called, `recoset` returns a zero value. If a bus error occurs subsequently, the kernel does a non-local jump so that the execution flow resumes again where `recoset` returns, this time with a non-zero value. `noreco` restores the default system behavior. An exception to this general scheme is the driver install entry point, discussed in Chapter 2, “Driver Structure”.

Probing for Devices

It is very common for the driver to test for the presence of a device during the `install` entry point. For this reason, LynxOS handles bus errors during execution of the install routine, thus relieving the driver of this responsibility. If a bus error occurs, the kernel does not return to the install routine from the bus error handler. The error is taken to mean that the device is not present and user tasks will not be permitted to open it. See “Install Entry Point” on page 13.

Device Access

The following sections contain platform-specific information about hardware device access from LynxOS. Each section contains memory map figures to illustrate the mapping of LynxOS virtual memory to the hardware device.

In general, the kernel has permissions to access the full range of virtual memory while the user processes have restricted access. Table 5-1 shows a generalization of this concept. Keep this in mind when viewing the memory maps.

Table 5-1: Virtual Memory Access to User Processes

LynxOS Virtual Memory Area	Permissions
OSBASE and above	Kernel only; no user access
SPECPAGE	Read-only to user
Kernel Stack	Read-only to user
Shared Memory	Depends on mapping
User Stack	Read-write to user
User Data	Read-write to user
User Text	Read-only to user

Device Access on x86 Systems

Refer to Figure 5-1 on page 67 and Figure 5-2 on page 68 for illustrations of LynxOS virtual memory on the x86.

Reading and Writing Device Registers

The majority of devices for x86 systems exist in the CPU's I/O space which is accessed with the *in* and *out* instructions. The file `<port_ops_x86.h>` contain macros that can be called to read and write device registers.

The memory on I/O devices that are in the 640 KB to 1 MB range can be directly accessed using the `PHYSBASE` offset. The constant `PHYSBASE` is defined in `<kernel.h>`. For example, to access I/O devices using the `PHYSBASE` offset:

```
/* RAMBASE: RAMbase address of Ethernet controller */
#define RAMBASE 0xCC000
unsigned long * vaddr;
vaddr = (unsigned long *) (PHYSBASE + RAMBASE);
```

Using permap

The function `permap` allows a driver to map a memory-mapped device into the kernel virtual memory map so that the device can be accessed from the driver. If the memory region in which the device resides is already mapped in, then it is not necessary to use `permap`. The virtual address `PHYSBASE` is always mapped to the physical address corresponding to the start of RAM. The size of the mapped region is equal to the system RAM size (up to 512 MB).

You must use `permap` to access devices that are outside of the pre-mapped region (640 KB to 1 MB). For example, in a VMEbus-based PC the VMEbus is often mapped in the high end of the physical address space, above 512 MB. The code to map this in using `permap` would be along the following lines:

```
/* VMEADDR: physical address where VMEbus appears */
#define VMEADDR  0x40000000

/* VMESIZE: window size onto VMEbus */
#define VMESIZE  0x00100000

/* vme_addr: virtual address 32 bit VMEbus accesses */
unsigned long *vme_vaddr;

vme_vaddr=(unsigned long *) permap
((long)VMEADDR,(long)VMESIZE);
```

The physical address passed to `permap` must be aligned on a page boundary. The size, in bytes, must be a multiple of `PAGESIZE`.

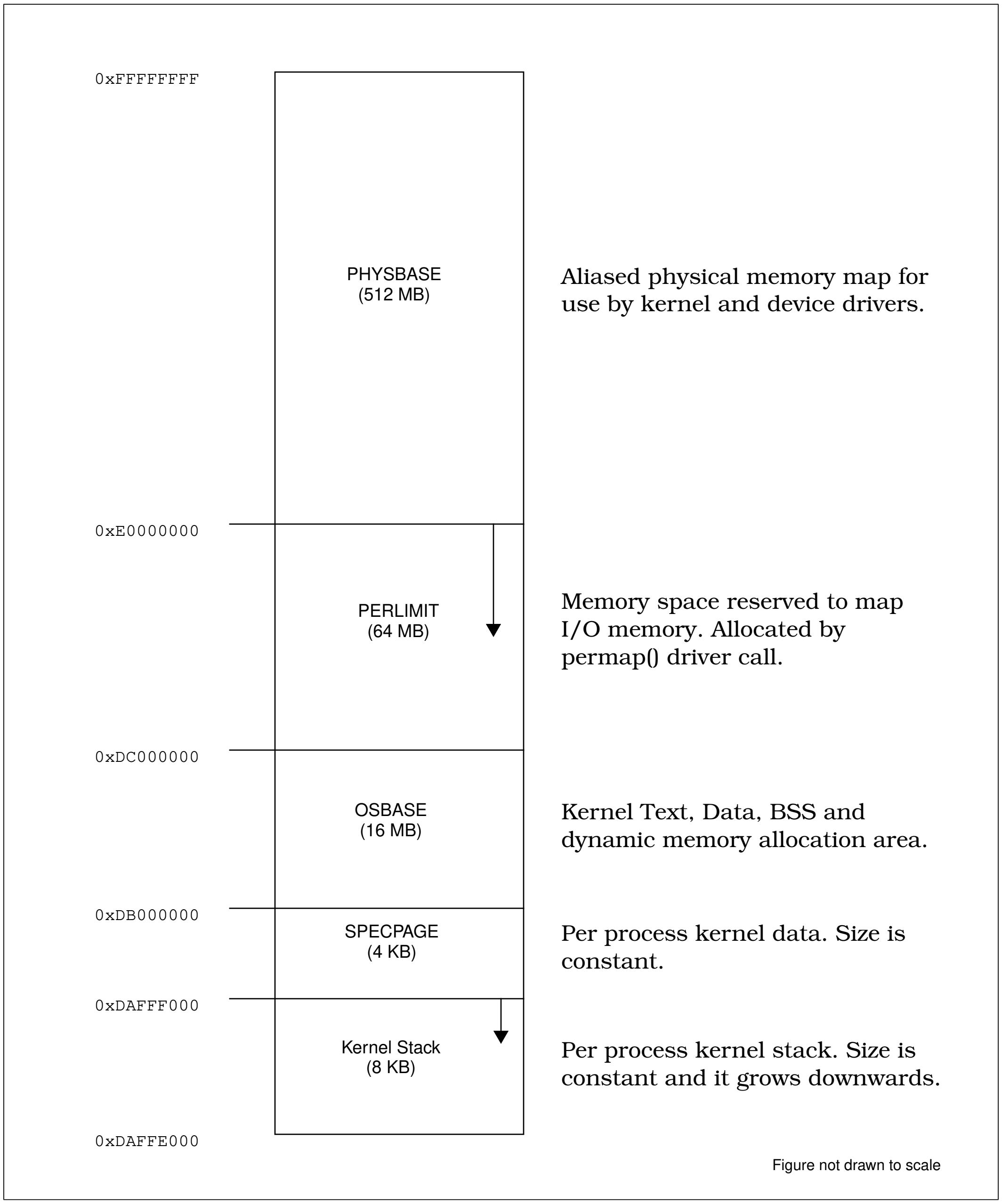

Figure 5-1: LynxOS x86 Kernel Access Virtual Memory Map

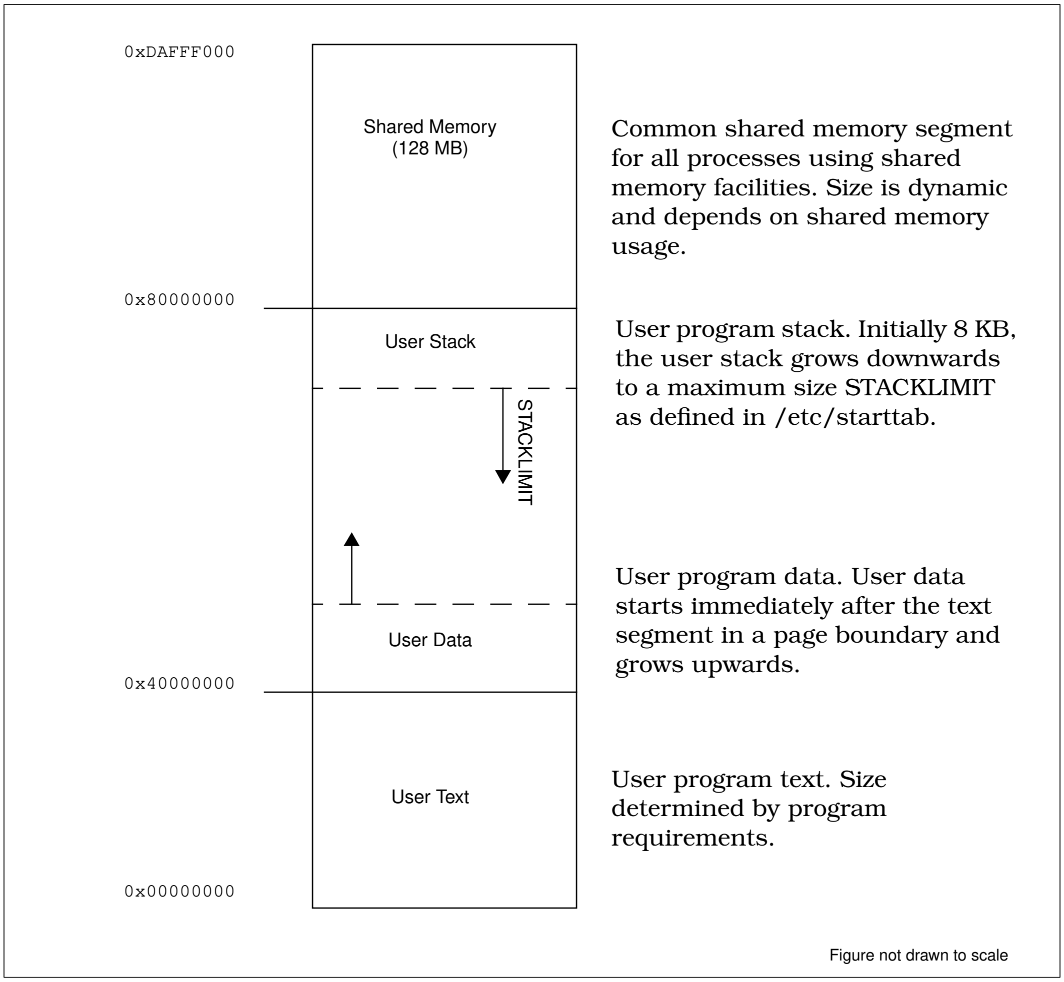

Figure 5-2: LynxOS x86 User Access Virtual Memory Map

Device Access on 68k Systems

Refer to the memory map diagrams (Figure 5-3 on page 71 and Figure 5-4 on page 72) for illustrations of the concepts discussed below. The constants PHYSBASE and VMEBASE are defined in <mem.h>.

VME A24/A32/D32

A 288 MB window in the A24/A32/D32 space is mapped in at virtual address PHYSBASE. As Figure 5-3 shows, the lower part of this virtual region is mapped to system RAM. So, in fact, the lower part of the VME address space is not accessible. The size of this inaccessible region is the same as the system RAM size. For example, on a system with 32 MB RAM, the VME window would be reduced to 256 MB, starting at VME address 0x2000000. For this reason, it is important to always configure devices in this VME address space with addresses above system RAM. The virtual address used to access a device is calculated simply by adding PHYSBASE to the physical VME address:

```
/* VME_PHYSADDR: device addr on VME bus */
#define VME_PHYSADDR    0x4000000

/* vme_virtaddr: virtual address to access 32-bit
device */
unsigned long *vme_virtaddr;

vme_virtaddr=(unsigned long *)
((int)VME_PHYSADDR+PHYSBASE);
```

VME A24/D16

The entire 16 MB of the A24/D16 space is mapped in at virtual address VMEBASE. The address used within a device driver to access a device in this space is calculated as follows:

```
vme_virtaddr = VMEBASE + vme_physaddr;
```

VME A32/D16 and A16/D16

LynxOS maps a 8 MB window into the upper part of the A32/D16 space at virtual addresses 0xFF000000–0xFF7FFFFF. The entire VME short IO space is mapped at virtual addresses 0xFFFF0000–0xFFFFFFFF. Both these address ranges fall within the region that LynxOS maps transparently. This means that the physical address is the same as the virtual address (there is no translation). Therefore, a driver can use the physical address directly to access a device in these spaces.

Accessing IP Slots

The IP interface specifies 3 address spaces—IO, ID and memory. Typically, IP modules are present in IO and ID space. On the 162, the IO and ID spaces for each IP slot are at fixed addresses (specified in the Motorola manual). LynxOS maps all virtual addresses in the range 0xFF000000–0xFFFFFFFF transparently i.e. the virtual address is the same as the physical address. The IO and ID spaces for IP slots are in this range so drivers can access them directly using the appropriate physical address. The memory space for IP slots is programmable (see Motorola documentation) but is always in the transparently mapped range so it too can be accessed directly from a driver.

Accessing SRAM

This is at a fixed address in the transparently mapped range (see Motorola documentation) so it is accessed using the physical address from a driver.

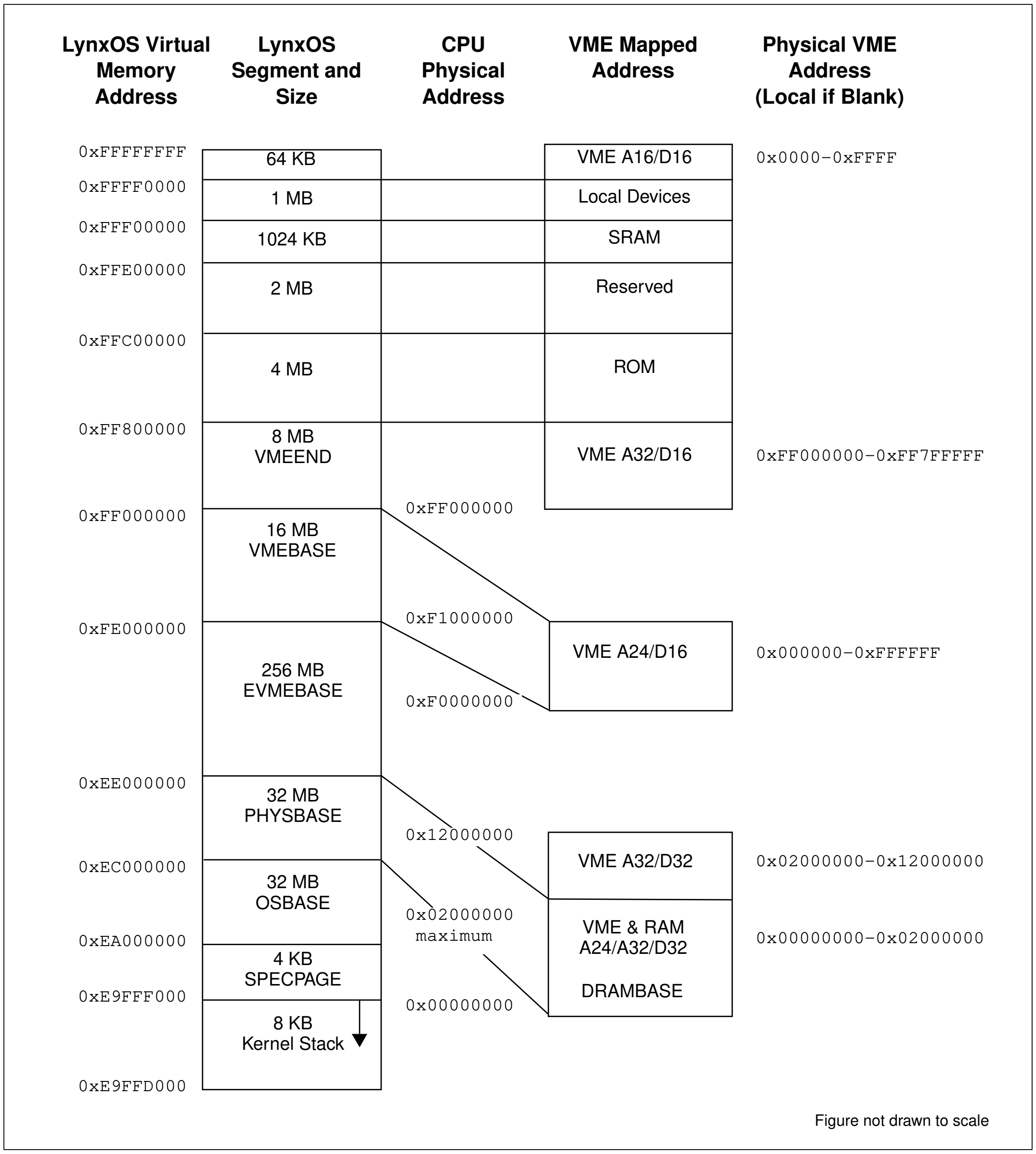

Figure 5-3: LynxOS 68k Kernel Access Virtual Memory Map

.

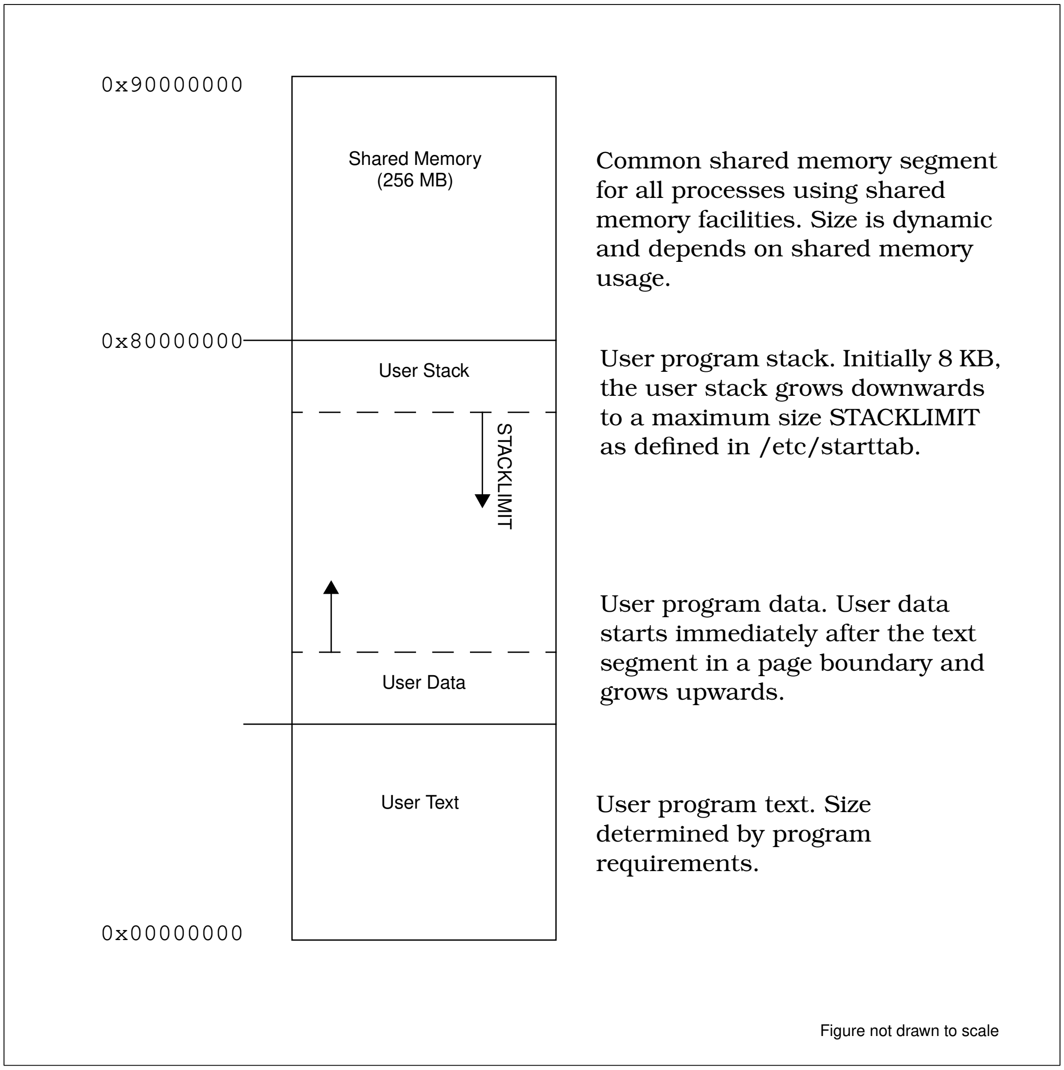

Figure 5-4: LynxOS 68k User Access Virtual Memory Map

Device Access on SPARC Systems

Refer to the memory map diagrams (Figure 5-5 on page 74 and Figure 5-6 on page 75) for illustrations of the concepts discussed below.

SBUS Details

Devices on the SBUS can be mapped using the function `allocate_device_space`. Constants for the physical addresses of the SBUS slots are defined in `<devspace.h>`.

```
#define BRD_OFFSET   0x800000
#define BRD_SIZE     0x100000
virt_addr = allocate_device_space ((long) (PV_SBUS1 +
BRD_OFFSET), BRD_SIZE);
```

The function returns `NULL` if space can not be allocated.

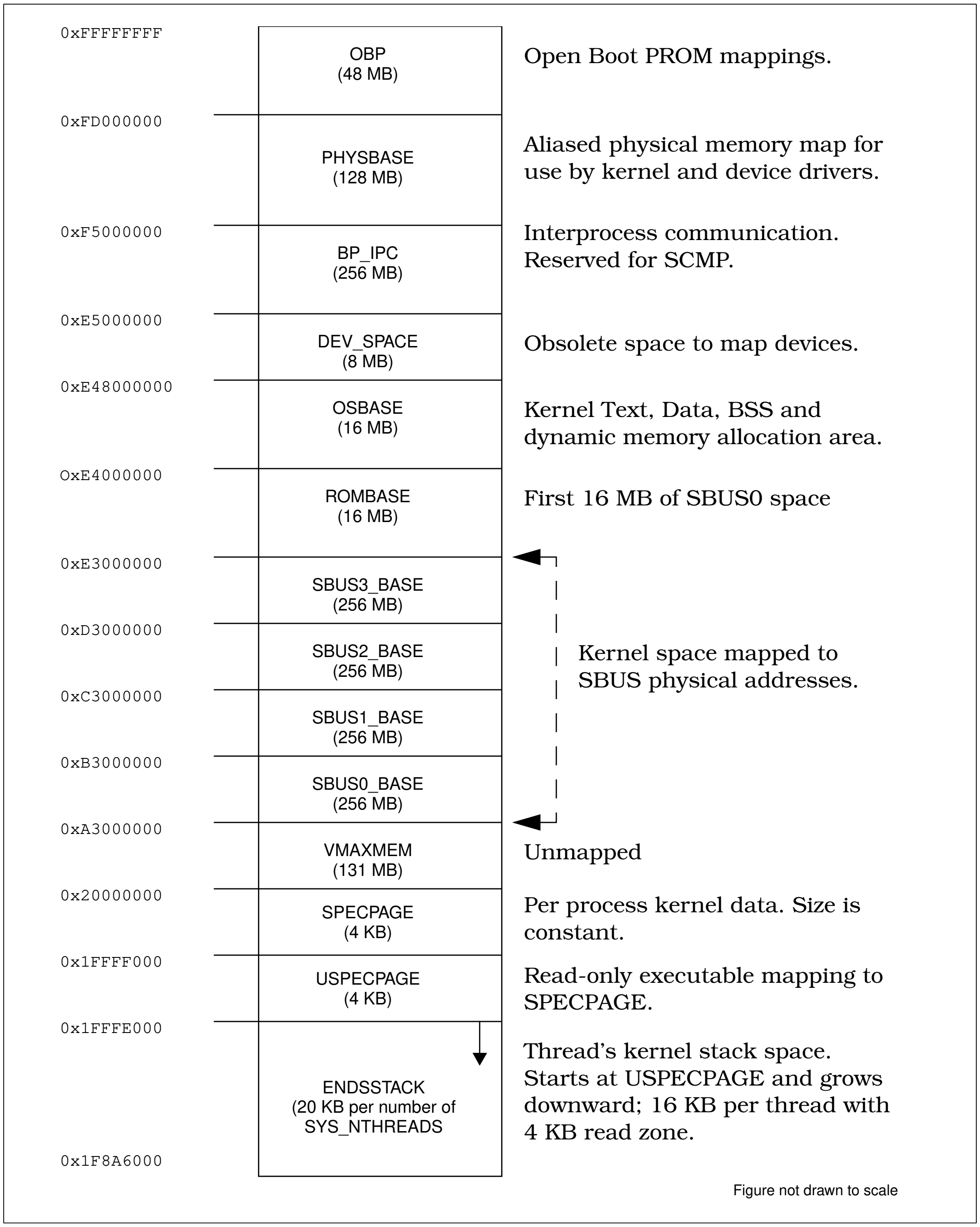

Figure 5-5: LynxOS SPARC Kernel Access Virtual Memory Map

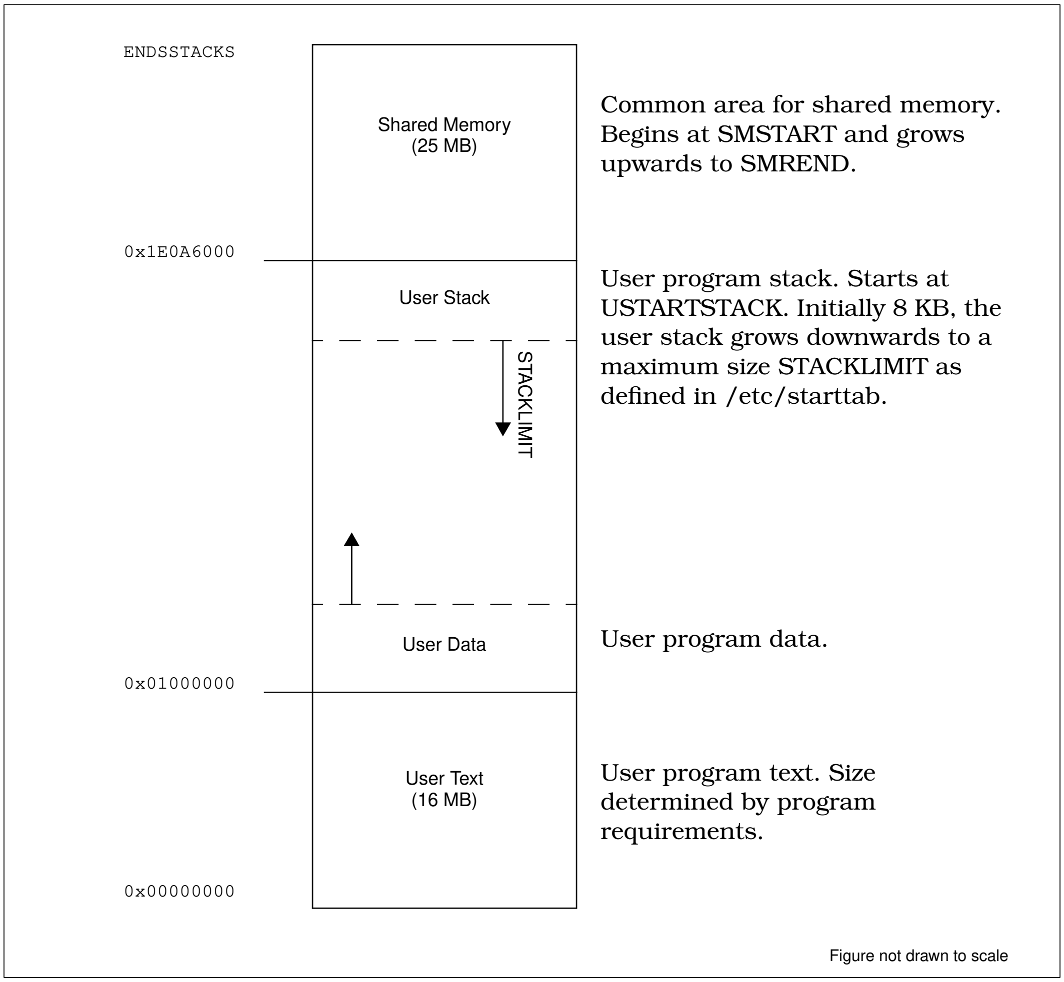

Figure 5-6: LynxOS SPARC User Access Virtual Memory Map

Device Access on PowerPC Systems

Refer to Figure 5-7 on page 80 and Figure 5-8 on page 81 for illustrations of LynxOS virtual memory on the PowerPC.

PCI and ISA Bus Access

The PowerPC reference platform contains a primary PCI bus and a secondary ISA bus for system I/O. All access to the ISA bus goes through the PCI bus and the PCI-to-ISA bridge hardware. On Motorola VME systems, the VME bus interfaces to the PCI bus through the VME-to-PCI bridge hardware. The PReP reference memory map defines all physical addresses above 2 GB to be directed to the PCI bus and all physical addresses less than 2 GB as memory access (see Figure 5-8). The PCI devices on the PCI bus are configured to claim address ranges. The ISA Bridge hardware claims all unclaimed PCI address. This is referred to as subtractive decoding.

The physical address of all ISA devices are mapped to 2GB+<default ISA address on x86> in a 64K address range. For example, the serial ports COM1 and COM2 reside at 0x3F8 and 0x2F8 on x86-based PCs. On the PReP reference hardware, COM1 and COM2 are mapped to 0X800003F8 and 0X800002F8. It is possible to think of ISA devices being shifted by 2 GB address and are memory mapped devices rather than I/O-mapped devices. To ensure that access to the I/O devices go as planned it is necessary to add the PowerPC `eieio` (enforce in-order execution of I/O) instruction between access to the device. A service function `eieio()` is provided by the LynxOS kernel. Also, if there is a necessity to ensure completion of all writes, the PowerPC instruction `sync` is available as a driver service call `sync()`.

Access to the PCI devices are based on the configuration of the individual devices. The configuration space of all the PCI devices are fixed by the architecture. The PCI devices are inherently little endian. LynxOS uses the PowerPC CPU in the default big endian mode, requiring that all access to the PCI device registers need to be byte swapped.

The CPU views the I/O space at a 2 GB offset in the 4 GB address space. Similarly the IO-controller views the memory at a offset of 2 GB. When doing DMA transfers, it is necessary to add 2 GB to the real memory address.

PCI Support Facilities

`pci_what_irq(int slot);`

Returns the IRQ level associated with the PCI slot *slot*. Slots have values between 0 and `PCI_NSLOTS`-1. On-board PCI devices are also assigned slots. The IRQ level associated with a slot is dependent on the platform. The IRQ bindings are established during kernel initialization from values defined in `<absolute.h>`.

`void pci_conf_write(int slot, int offset, int value);`

Writes the value *value* into the configuration space of the PCI device present at slot *slot*. The value is written at an offset *offset* from the start of the configuration space. *offset* must be a word aligned quantity. Returns -1 if the slot or offset is not valid.

`int pci_conf_read(int slot, int offset);`

Reads the value at offset *offset* in the configuration space of the PCI device at slot *slot*. The read value is returned by the function. *offset* must be a word aligned quantity. Returns -1 in case the slot is not valid, or the offset is not valid.

`int pci_probe_start();`

Returns the first unclaimed PCI slot. Slots can be claimed by drivers so that other drivers will not get the same slot. Returns -1 in case there are no unclaimed slots.

`int pci_probe_next(int slot);`

Returns the next unclaimed slot starting from the given *slot*. Returns -1 if there are no unclaimed slots.

`pci_claim_slot(int slot);`

Makes the *slot* unavailable to other probes. Returns -1 if *slot* is invalid.

`pci_unclaim_slot(int slot);`

Makes *slot* available to other probes. Returns -1 if *slot* is invalid.

`alloc_dev_space(int spaceid, int size, int *pap, int *vap)`

Allocates device space from `PCI_MEM`, `IO_MEM` or `PCI_IO`. Request size is in *size* and the physical and virtual address of the allocated space are returned in *pap* and *vap*.

`PCI_CONF_SLOT(slot)`

Macro which returns the virtual address of the configuration space for device at PCI slot *slot*.

Example

The following code segment is a sample PCI setup code for a device driver.

```
llscsipci_attach(s)
register scsistat *s;
{
    int slot = pci_probe_start();
    int vdid;

    for (; slot >= 0; slot = pci_probe_next(slot)) {
        vdid = pci_conf_read(slot, 0);
        if (vdid == NCR53C810 || vdid == NCR53C825)
            break;
    }
    if (slot < 0)
        return 0;
    s->vector = pci_what_irq(slot);
    if (s->vector < 0)
        return 0;
    s->slot = slot;
    s->IOCport = (int *)PCI_CONF_SLOT(slot);
    if (!alloc_dev_space(PCI_MEM, PAGESIZE,
        &s->NCRportpa, &s->NCRport))
        return 0;
    pci_claim_slot(slot);

    return vdid;
}
```

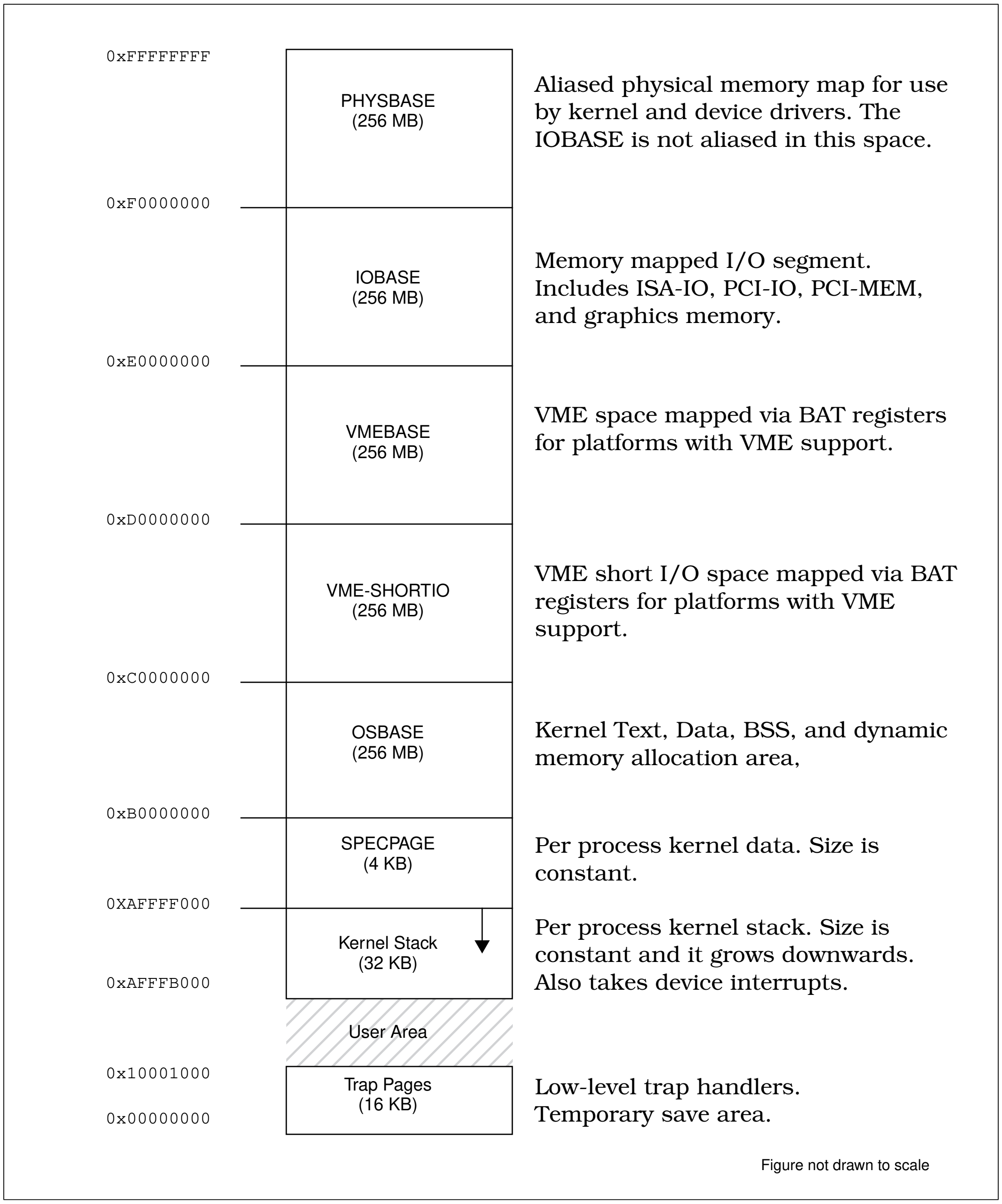

Figure 5-7: LynxOS PowerPC Kernel Access Virtual Memory Map

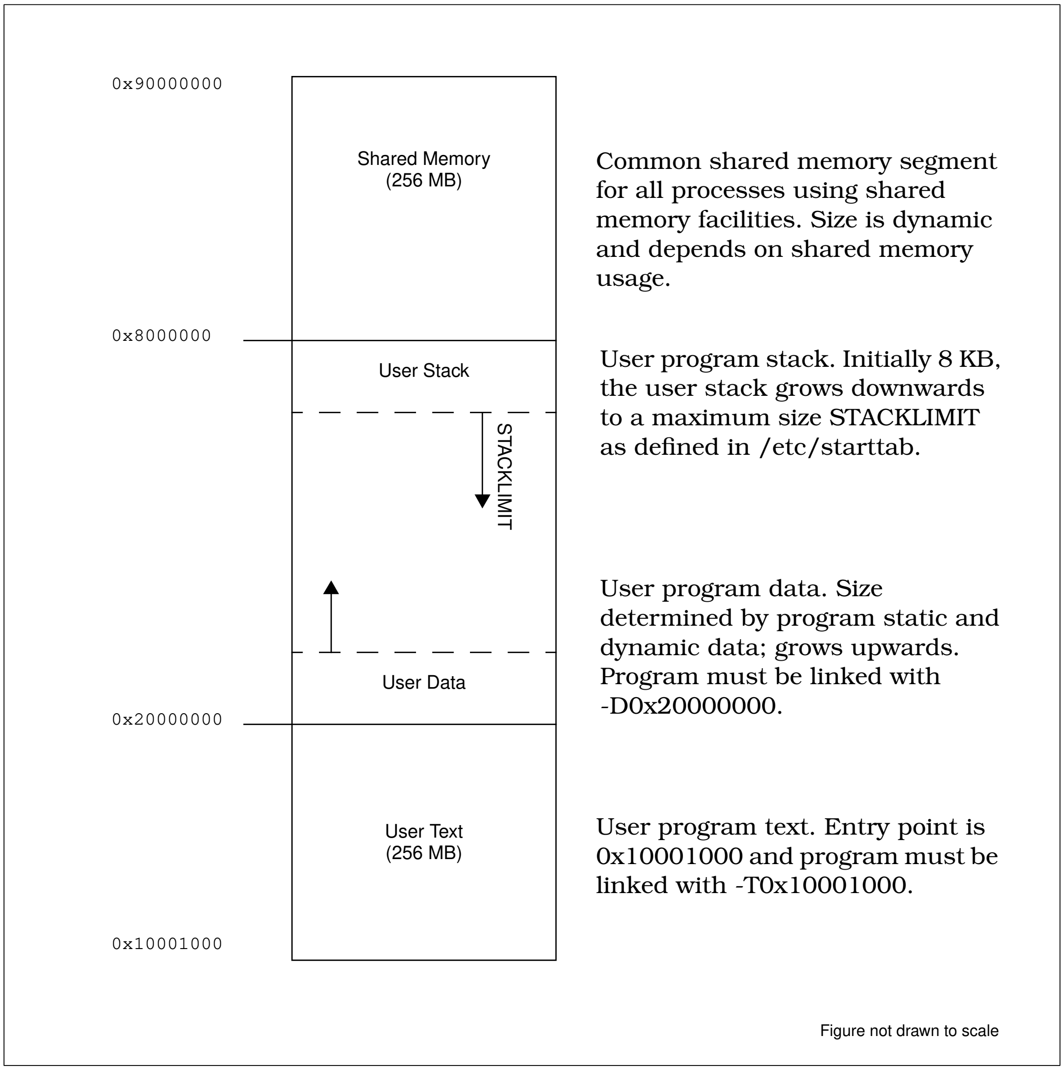

Figure 5-8: LynxOS PowerPC User Access Virtual Memory Map

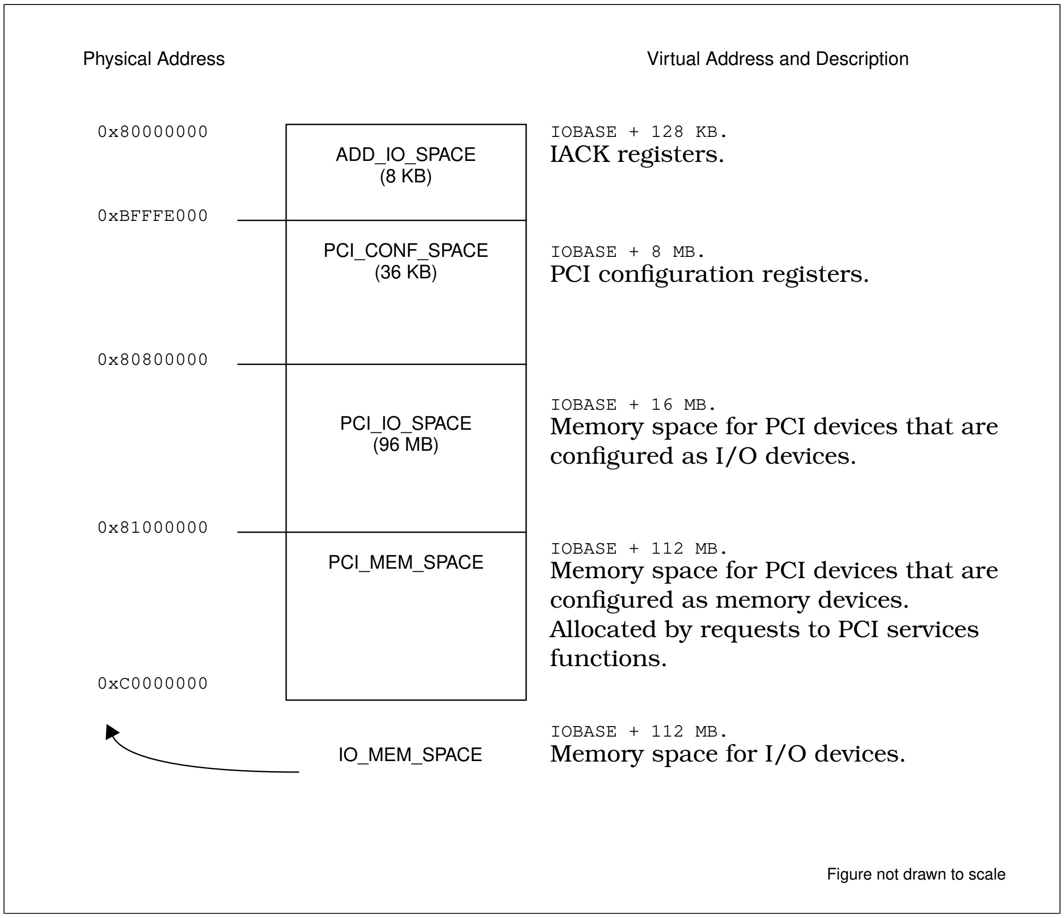

Figure 5-9: Details of IOBASE Segments

CHAPTER 6

Elementary Device Drivers

This chapter discusses the details of some simple device drivers that do everything in a polled fashion.They are efficient in the sense that they do not interfere with the real-time response of the system.

It is useful to see a comparison between these drivers and their more efficient interrupt-based and thread-based implementation in order to understand the performance advantages of one over the other.

Character Device Drivers

An elementary printer driver is shown below. This driver does not use interrupts. Instead, it polls the port to check if the previous character was put on the port.

Device Information Definition

```
/* ptrinfo.h */

struct ptrinfo {
    int port;
};
```

The device information definition for this printer driver only contains the port address definition.

Device Information Declaration

```
/* ptrinfo.c */

#include "ptrinfo.h"

struct ptrinfo ptrinfo0 = {
    /* port */ 0x378
};
main()
{
    write(1, &ptrinfo0, sizeof(struct ptrinfo));
}
```

The device information declaration for the above device information definition gives the physical address for the printer port which is 0x378 for an x86. The program `ptrinfo.c` is compiled and executed to create a data file to be passed to the `install` routine during dynamic installation. For a pre-linked device, the `main` routine is not needed.

Declaration for ioctl

```
/* ptrioctl.h */

#define PTRSTATUS 500

struct ptrstatus {
    int chars;     /* characters printed */
    int lines;     /* lines printed      */
};
```

The `ioctl` routine in this driver gives the number of characters and lines printed out so far. Thus, the user can issue the `ioctl` system call with the appropriate command and pointer to the above-defined structure; the number of lines and characters printed so far are returned to the user.

Driver Source Code

```
/* ptrdrvr.c */

#include <kernel.h>
#include <mem.h>
#include <file.h>
#include <errno.h>
#include <ioctl.h>
#include "ptrinfo.h"
#include "ptrioctl.h"

/* ports */
#define PP_DATA     0      /* data port offset */
#define PP_STATUS   1      /* status port offset */
#define PP_CONTROL  2      /* control port offset */
```

The printer port usually has 3 sub-ports: a data port where the data is written, a status port to inquire about the previous operation, and a control port to which to send a strobe to latch the data to the printer.

```
/* status bits */

#define PP_BUSY   0x80     /* printer busy */
#define PP_PE     0x20     /* out of paper */
#define PP_SLCT   0x10     /* printer is selected */
#define PP_ERROR  0x08    /* printer detected error */

/* control bits */
#define PP_IENABLE  0x10   /* interrupt enable */
#define PP_SLCTIN   0x08   /* select printer */
#define PP_INIT     0x04   /* start printer */
#define PP_AUTOLF   0x02   /* auto line feed */
#define PP_STROBE   0x01   /* strobe printer */
```

Statics Structure

```
struct ptrstatics {
    int port;
    int opened;
    int write_sem;
    int open_sem;
    int chars;
    int lines;
};
```

The statics structure is defined as shown. The port address passed through the device information structure is stored in the structure. The variable opened is a flag indicating whether or not the device has already been opened by another process. The variable `write_sem` is a semaphore used to prevent more than one process from writing to the printer port. And finally, the chars and lines variables store the number of characters and lines printed out so far.

```
static port_in(), port_out();

asm {
        port_in:            /* byte = port_in(port) */
            mov EAX, 0
            mov EDX, 4[ESP]
            in  AL, DX
            ret
        port_out:           /* port_out(byte, port) */
            mov EDX, 8[ESP]
            mov EAX, 4[ESP]
            out DX, AL
            ret
}
```

These two routines (`port_in()` and `port_out()`) can be used to read a byte of a port or write a byte onto a port. `port_in` reads a byte from the specified port and returns the value. `port_out` puts the byte specified onto the given port.

Many DOS compilers have these instructions built in. The above assembly routines perform the same function as those built-in functions.

Install Routine

```
char *ptrinstall(info)
struct ptrinfo *info;
{
    struct ptrstatics *s;
        int i;
                        /* probe for the printer */
        port_out(1, info->port+PP_DATA);
        if (port_in(info->port+PP_DATA) != 1)
            return (struct ptrstatics *) SYSERR;
        if (!(s = (struct ptrstatics *)
                 sysbrk(sizeof *s)))
            return (struct ptrstatics *) SYSERR;
                        /* initialize statics */
        s->port = info->port;
        s->opened = 0;
        s->write_sem = -1;
        s->open_sem = -1;
                        /* initialize printer */
        port_out(PP_SLCTIN, s->port+PP_CONTROL);
        for (i = 0; i < 100; i++) ;
        port_out(PP_SLCTIN | PP_INIT,
               s->port+PP_CONTROL);
        return (char *) s;
}
```

The `install` routine basically checks for the existence of the printer. The data is written into the port and read back immediately. If the data is not the same, the printer is non-existent. Once the presence of the printer is confirmed, the statics structure is allocated and initialized. Finally, an initialization sequence is sent to the printer.

Notice that there is a timing loop between the two `port_out`'s. This is required for the transmitted data to be latched properly. The pointer to the statics structure is then returned.

Uninstall Routine

```
void ptruninstall(s)
struct ptrstatics *s;
{
    sysfree(s, (long)sizeof *s);
}
```

The `uninstall` routine frees the memory allocated for the statics structure in the `install` routine.

Open Routine

```
int ptropen(s, devno, f)
struct ptrstatics *s;
int devno;
struct file *f;
{
    if (f->access_mode & FREAD) {
        pseterr(EINVAL);
        return SYSERR;
    }
    if (!s->opened) {
        s->opened = 1;
        s->chars = s->lines = 0;
    }
    return OK;
}
```

The `open` routine first checks whether or not the access mode is read. If it is anything but read, an error is signalled. On the first open, the chars and lines variables are initialized to 0.

Close Routine

```
int ptrclose(s, f)
struct ptrstatics *s;
struct file *f;
{
    s->opened = 0;
    return OK;
}
```

The `close` routine sets the value of the opened variable to zero to indicate that the device is no longer used by any process.

Auxiliary Routine

```
void write_char(s, c)
struct ptrstatics *s;
char c;
{
    while (!(port_in(s->port+PP_STATUS) & PP_BUSY)) ;
    port_out(c, s->port+PP_DATA);
    port_out(PP_SLCTIN | PP_INIT | PP_STROBE,
          s->port+PP_CONTROL);
    port_out(PP_SLCTIN | PP_INIT, s->port+PP_CONTROL);
    s->chars++;
}
```

This is the routine which actually puts a character in the printer port. The `while` loop checks if the port is busy by reading the status port and checking for the busy bit. If the port is not busy, the character is put into the data port. After this, the byte is latched into the printer by sending the port a high-low strobe. The number of characters printed is then incremented.

Write Routine

```
int ptrwrite(s, f, buff, count)
struct ptrstatics *s;
struct file *f;
char *buff;
int count;
{
    int i = count;
    char c;

    swait(&s->write_sem, SEM_SIGRETRY);
    while (i--) {
        c = *buff++;
        write_char(s, c);
        if (c == '\n') {
            write_char(s, '\r');
            s->lines++;
        }
    }
    ssignal(&s->write_sem);
    return count;
}
```

The `write` routine has a `swait` at the beginning to ensure that write requests of one task do not get mixed up with those of another. The character is retrieved from the buffer and the `write_char` routine is invoked to print the character. If a newline is found, a CR-LF (carriage return–line feed) sequence is sent to the printer and the number of lines is incremented.

Ioctl Routine

```
int ptrioctl(s, f, command, arg)
struct ptrstatics *s;
struct file *f;
int command;
char *arg;
{
    switch (command) {
        case PTRSTATUS:
         if (wbounds(arg) < sizeof(struct ptrstatus))
{
                    pseterr(EFAULT);
                    return SYSERR;
            }
          ((struct ptrstatus *)arg)->chars = s->chars;
          ((struct ptrstatus *)arg)->lines = s->lines;
            break;

        case FIOPRIO:

        case FIOASYNC:
        break;

        default:
            pseterr(EINVAL);
            return SYSERR;
    }
    return OK;
}
```

The `ioctl` routine operates based on the command issued by the user. If the command is `PTRSTATUS`, the number of lines and characters printed out is returned to the user. The routine `wbounds()` checks if the user's pointer is valid. (See the *LynxOS Reference Manuals* under "Driver Service Calls"). If the pointer is valid, the two variable values are copied to the user's buffer.

Dynamic Installation

```
#include <dldd.h>
extern int ionull ();
static struct dldd entry_points = {
    ptropen, ptrclose, ionull, ptrwrite,
    ptrselect, ptrioctl,
    ptrinstall, ptruninstall, (char *) 0
};
```

This is the declaration for dynamic installation. There is no read entry point in the printer driver so the kernel fucntion `ionull` is declared in the structure.

CHAPTER 7

Interrupt Handling

Interrupts are external hardware exception conditions which are delivered to the processor indicating the occurrence of a specific event. Some of the things for which they are useful are listed below:

- Indication of the completion of an operation.

  For example, an interrupt could be generated indicating the completion of a DMA (Direct Memory Access) transfer. The device driver would give a command to the DMA controller to transfer a block of data and set the vector for the interrupt generated by the controller to a specific driver function. This, in turn, would signal a semaphore to wake up any system or user or threads waiting on the completion of the DMA transfer.
- Data availability.

  The availability of data at a port is often indicated by an interrupt. A tty driver receives an interrupt when a character is ready to be read from the port.
- Device ready for a command.

  A printer generates an interrupt when it has printed a character and is ready to print the next character.

LynxOS Interrupt Handlers

Interrupt handlers in LynxOS are specified in the `install` or `open` entry points and are cleared in the `uninstall` or `close` entry points. These interrupt handlers run before any other kernel or application processing is completed.

Since interrupt handlers have the highest run priority, the minimization of the length of each interrupt service routine is paramount. This leads to better interrupt response time and task response time. Note that all interrupts are disabled during the execution of an interrupt handler.

Interrupt handlers in LynxOS are declared and reset using the functions `iointset()` and `iointclr()`. (See the *LynxOS Reference Manuals* under "Driver Service Calls"). Interrupt handlers and other driver calls cannot directly use application virtual addresses. The application virtual addresses must be translated to kernel virtual addresses before they can be accessed by driver routines.

On x86 platforms, the interrupt vector (0–15) must be offset by 32:

```
iointset (irq + 32, intr_handler, s);
iointclr (irq + 32);
```

On 68k platforms, the interrupt vector (64–255) must be multiplied by 4:

```
iointset (vec * 4, intr_handler, s)
iointclr (vec * 4, intr_handler);
```

General Format

The general format of an interrupt-based device driver under LynxOS is as follows:

- The general driver entry points are referred to as the top-half of the device driver.
- The interrupt handler is known as the bottom-half of the driver.

Use of Queues

Queues are often used to communicate between the top and bottom halves. Examples of the use of queues to communicate between entry points and interrupt handlers follow:

- For communication from the `write()` entry point to the interrupt handler.

  A counting semaphore, initialized to the size of the queue, tracks the free space in the `write()` entry point. The `swait()` routine is called and if space is available in the queue a character is enqueued. The interrupt handler subsequently dequeues the character and signals the semaphore using `ssignal()`.

- For communication from the interrupt handler to the `read()` entry point.

  The semaphore in the `read()` entry point tracks data availability in the queue. The `swait()` routine blocks until data is available in the queue. The interrupt handler enqueues the data, signaling the semaphore that data is available, if queue space is available. If queue space is unavailable, an error flag is set.

Typical Structure of Interrupt-Based Driver

```
device_read()
{
    swait(&receive_data_available,SEM_SIGIGNORE);
    disable();
    dequeue_receive_data();
    restore();
}
device_write()
{
    disable();
    swait(&space_on_queue_available,SEM_SIGIGNORE);
    enqueue_send_data();
    if (no_interrupt_pending)
        output_data();
    restore();
}
interrupt_handler()
{
    if (data_received) {
        enqueue_receive_data();
        ssignal(&receive_data_available);
    }
    else {
        if (dequeue_send_data())
            output_data();
        else no_interrupt_pending = 1;
    }
}
```

Example

What follows is an example of an interrupt-based printer device driver following the format specified in Chapter 6, "Elementary Device Drivers".

Device Information Definition

```
/* ptrinfo.h */
struct ptrinfo {
    int port;
    int irq;
    int qlen;
};
```

The device information definition has three variables associated with the interrupting driver. The `port` variable is the port number for the printer; `irq` is the interrupt line on which the printer interrupts the system; and `qlen` is the length of the queue used for communication between the top and bottom halves of the driver.

Device Information Declaration

```
/* ptrinfo.c */

#include "ptrinfo.h"

struct ptrinfo ptrinfo0 = {
            /* port */ 0x378,
            /* irq  */ 7,
            /* qlen */ 100
};

main()
{
    write(1, &ptrinfo0, sizeof(struct ptrinfo));
}
```

The device information declaration for the above device information definition gives the port address for the printer `port` which is `0x378` for a x86. The `irq` line on which the printer interrupts is `7` and the queue length is `100`. The program `ptrinfo.c` is compiled and executed to create a data file to be passed to the `install` routine during dynamic installation.

Declaration for Ioctl

```
/* ptrioctl.h */

#define PTRSTATUS 500

struct ptrstatus {
    int chars;                /* characters printed */
    int lines;                /* lines printed */
};
```

The `ioctl` routine in this driver gives the number of characters and lines printed out so far. Thus, the user can issue the `ioctl` system call with the appropriate command and pointer to the above-defined structure; the number of lines and characters printed so far are returned to the user.

Driver Source Code

```
/* ptrdrvr.c */

#include <kernel.h>
#include <mem.h>
#include <file.h>
#include <errno.h>
#include <ioctl.h>
#include "ptrinfo.h"
#include "ptrioctl.h"

/* ports */
#define PP_DATA       0       /* data port offset */
#define PP_STATUS     1       /* status port offset */
#define PP_CONTROL    2       /* control port offset */

/* status bits */
#define PP_BUSY       0x80    /* printer busy */
#define PP_PE         0x20    /* out of paper */
#define PP_SLCT       0x10    /* printer is selected  */
#define PP_ERROR      0x08    /* printer detected error */

/* control bits */
#define PP_IENABLE    0x10    /* interrupt enable  */
#define PP_SLCTIN     0x08    /* select printer */
#define PP_INIT       0x04    /* start printer */
#define PP_AUTOLF     0x02    /* auto line feed */
```

```
#define PP_STROBE     0x01    /* strobe printer */

typedef unsigned short ptype;
```

Statics Structure

```
struct ptrstatics {
    ptype datap;        /* data port address   */
    ptype controlp;     /* cntrl port address  */
    char control;       /* control bits        */
    int irq;            /* IRQ number          */
    int closing;        /* closing device      */
    int close_sem;      /* semaphore for close */
    int expecting;      /* expecting an int.?  */
    int nextnl;         /* output a '\r' next? */
    int chars;          /* printed since open  */
    int lines;          /* printed since open  */
    int qlen;           /* characters in queue */
    char *q;            /* base queue address  */
    int head;           /* head of queue       */
    int tail;           /* tail of queue       */
    int qdata;          /* data in the queue   */
    int free_sem;       /* free queue space    */
};
```

The statics structure for the interrupt handling printer driver is shown above. The `irq` number is copied from the device information definition as is the queue length `qlen`. The port address passed through the device information definition is stored in the structure.

`chars` and `lines` store the number of characters and lines printed out so far. `q` is the base address of the queue. `head` and `tail` keep track of the data in the queue.

`close_sem` is a semaphore used for ensuring that the output queue is fully drained before the device is closed. `expecting` is initially reset to indicate that the first character has to be output before an interrupt is received by the system. `nextnl` is used for mapping `\n` to `\r\n` (i.e., the carriage return line feed sequence).

```
static port_in(), port_out();

asm {
    port_in:                /* byte = port_in(port) */
        mov EAX, 0
        mov EDX, 4[ESP]
        in  AL, DX
        ret
    port_out:               /* port_out(byte, port) */
        mov EDX, 8[ESP]
        mov EAX, 4[ESP]
        out DX, AL
        ret
}
```

These two routines can be used to read a byte from a port or write a byte into a port. `port_in` reads a byte from the specified port and returns the value. `port_out` puts the byte specified onto the given port.

Install Entry Point

```
#define PERR (struct ptrstatics *) SYSERR

char *ptrinstall(info)
struct ptrinfo *info;
{
    struct ptrstatics *s;
    extern void ptrint();
    int i;

                           /* probe for the printer */
    port_out(1, info->port+PP_DATA);
    if (port_in(info->port+PP_DATA) != 1) {
        return (char *) PERR;
    }

    if (!(s = (struct ptrstatics *)sysbrk
                ((long)sizeof *s)))
        return (char *) PERR;
    if (!(s->q = sysbrk((long)info->qlen))) {
        sysfree(s, (long)sizeof *s);
        return (char *) SYSERR;
    }

                              /* initialize statics */
    s->datap = info->port + PP_DATA;
    s->controlp = info->port + PP_CONTROL;
    s->control = PP_SLCTIN | PP_INIT;
    s->irq = info->irq;
    s->expecting = 0;
    s->closing = s->close_sem = 0;
    s->nextnl = 0;
    s->free_sem = s->qlen = info->qlen;
    s->qdata = s->head = s->tail = 0;
    s->lines = s->chars = 0;

                              /* initialize printer */
    iointset(32+s->irq, ptrint, s);
    port_out(PP_SLCTIN, s->controlp);
    for (i = 0; i < 100; i++) ;
    port_out(s->control, s->controlp);

    return (char *) s;
}
```

The `install` routine checks for the existence of the printer. The data is written onto the port and read back immediately. If the data is not the same, the printer is non-existent. Once the presence of the printer is confirmed, the statics structure and the queue are allocated. The fields within the structure are initialized.

`free_sem` is initialized to the queue length and the `irq` number is copied into the statics data structure. The routine `iointset()` is called to initialize a interrupt handler for the given interrupt vector. (The offset of 32 is added to the `irq` number before passing it to the routine on the x86.) Finally, an initialization sequence is sent to the printer.

Notice that there is a timing loop between the two `port_out`'s. This is required for the transmitted data to be latched properly. The pointer to the statics structure is returned.

Uninstall Entry Point

```
void ptruninstall(s)
struct ptrstatics *s;
{
    iointclr(32+s->irq);
    sysfree(s->q, (long)s->qlen);
    sysfree(s, (long)sizeof *s);
}
```

The `uninstall` entry point clears the interrupt vector by using the `iointclr()` routine. `uninstall` frees the memory associated with the queue and the statics structure.

Open Entry Point

```
int ptropen(s, devno, f)
struct ptrstatics *s;
int devno;
struct file *f;
{
    if (f->access_mode & FREAD) {
        pseterr(EINVAL);
        return SYSERR;
    }
    return OK;
}
```

The `open` entry point checks for the access mode of the device and returns an error if the application has tried to open it in read mode.

Close Entry Point

```
int ptrclose(s, f)
struct ptrstatics *s;
struct file *f;
{
    int ps;

    disable(ps);
    if (s->expecting) {
        s->closing = 1;
        restore(ps);
        swait(&s->close_sem, SEM_SIGIGNORE);
        s->closing = 0;
    } else {
        restore(ps);
    }
    s->lines = s->chars = 0;
    return OK;
}
```

The `close` entry point ensures that the characters to be output are complete before the device is closed. It checks whether `s->expecting` is `1`. `1` indicates that there are characters present in the queue. If this is true it sets the `s->closing` flag and waits in the `swait()` routine for the interrupt handler to signal that all characters are output and the device can be closed. It also resets the `chars` and `lines` fields in the statics structure definition.

Send Routine

```
/*  assumes:
**    data in the queue
**    interrupts disabled
*/
void send(s)
struct ptrstatics *s;
{
    char c;

    if (s->nextnl) {
        c = '\r';
        s->nextnl = 0;
        s->lines++;
    } else {
        c = s->q[s->head++];
        s->head %= s->qlen;
        s->qdata--;
        ssignal(&s->free_sem);
        s->nextnl = c == '\n';
        s->chars++;
    }
    port_out(c, s->datap);
    port_out(s->control | PP_STROBE, s->controlp);
    port_out(s->control | PP_IENABLE, s->controlp);
}
```

This routine outputs a character into the port by dequeuing from the queue. If a newline is found it puts out a `\r` and resets the `nextnl` flag. If not, it dequeues from the head of the queue, adjusts the head pointer to wrap around, and signals the semaphore while keeping track of the free space in the queue. This routine then increments the number of characters `s->chars`.

The routine then actually puts a character onto the printer port. The while loop to check the status port is no longer necessary because the interrupt signifies that it is safe to write. The character is just put onto the data port. After this, to latch the byte onto the printer, a high-low strobe is put onto the control port.

Interrupt Handler

```
void ptrint(s)
struct ptrstatics *s;
{
    if (s->qdata || s->nextnl) {
        send(s);
    } else {
        s->expecting = 0;
        if (s->closing) ssignal(&s->close_sem);
                         /* disable ptr interrupts: */
        port_out(s->control, s->controlp);
    }
}
```

If the queue has data in it, or if a newline is indicated, the routine `send`(s) is called to output the character onto the port. If not, it indicates that the queue is empty and thus `s->expecting` is set to zero. Further, if the `closing` flag is set, it indicates that the `close` routine is in an `swait()`. Thus, a `ssignal` routine is called to awaken the semaphore. Finally, an initialization sequence is sent to the control port.

Write Entry Point

```
int ptrwrite(s, f, buff, count)
struct ptrstatics *s;
struct file *f;
char *buff;
int count;
{
    int i = count, ps;
    while (i--) {
        swait(&s->free_sem, SEM_SIGRETRY);
        disable(ps);
        s->q[s->tail++] = *buff++;
        s->tail %= s->qlen;
        s->qdata++;
        if (!s->expecting) {
            send(s);
            s->expecting = 1;
        }
        restore(ps);
    }
    return count;
}
```

The `write` entry point has a loop for `count` characters to be output onto the queue. The `swait()` inside the loop tracks the free space in the queue. If there is space in the queue a character is queued.

The `disable()` and `restore()` routines provide protection for the critical region of code which is used by the interrupt handler. The character is queued to the tail of the queue. The `qdata` variable (which keeps track of the number of characters in the queue) is incremented. If `expecting` is zero, the first character on the queue is sent to the port and `expecting` is set to one.

ioctl Entry Point

```
int ptrioctl(s, f, command, arg)
struct ptrstatics *s;
struct file *f;
int command;
char *arg;
{
    switch (command) {
        case PTRSTATUS:
          if (wbounds(arg) < sizeof(struct ptrstatus))
{
                pseterr(EFAULT);
                return SYSERR;
            }
           ((struct ptrstatus *)arg)->chars = s->chars;
           ((struct ptrstatus *)arg)->lines = s->lines;
            break;

        case FIOPRIO:
        case FIOASYNC:
            break;

        default:
            pseterr(EINVAL);
            return SYSERR;
    }
    return OK;
}
```

The `ioctl` entry point handles the case for `PTRSTATUS`. The user can invoke this from an application program to determine the number of characters which are actually output onto the port. `arg` is the pointer to the user buffer. The check by `wbounds` confirms that the user pointer has writable memory allocated to it.

CHAPTER 8

Kernel Threads & Priority Tracking

To off-load processing from interrupt-based sections of a device driver, LynxOS offers a feature known as *kernel threads*, also referred to as system threads. Kernel threads are defined as independently schedulable entities which reside in the kernel's virtual address space. They closely resemble processes but do not have the memory overhead associated with processes.

Although kernel threads have independent stack and register areas, the kernel threads share both text and data segments with the kernel. Each kernel thread has a priority associated with it which is used by the operating system to schedule it. Kernel threads can be used to improve the interrupt and task response times considerably. Thus, they are often used in device drivers.

Priority tracking is the method used to dynamically determine the kernel thread's priority. The kernel thread assumes the same priority as the highest-priority application which it is currently servicing.

Device Drivers in LynxOS

Device drivers form an important part of any operating system, but even more so in a real-time operating system such as LynxOS. The impact of the device driver performance on overall system performance is considerable. Since it is imperative for the operating system to provide deterministic response time to real-world events, device drivers must be designed with determinism in mind.

Some of the important components of real-time response are described in the following sections.

Interrupt Latency

Interrupt latency is the time taken for the system to acknowledge a hardware interrupt. This time is measured from when the hardware raises an interrupt to when the system starts executing the first instruction of the interrupt routine (in the case of LynxOS this routine is the interrupt dispatcher). This time is dependent on the interrupt hardware design of the system and the longest time interrupts are disabled in the kernel or device drivers.

Interrupt Dispatch Time

Interrupt dispatch time is the time taken for the system to recognize the interrupt and begin executing the first instruction of the interrupt handler. Included in this time is the latency of the LynxOS interrupt dispatcher (usually negligible).

Driver Response Time

The *driver response time* is the sum of the interrupt latency and the interrupt dispatch time. This is also known as the interrupt response time.

Task Response Time

The *task response time* is the time taken by the operating system to begin running the first instruction of an application task after an interrupt has been received that makes the application ready to run. This figure is the total of:

- The driver response time (including the delays imposed by additional interrupts)
- The longest preemption time
- The context switch time
- The scheduling time
- The system call return time

Only the driver response time and the preemption time are under the control of the device driver writer. The other times depend on the implementation of LynxOS on the platform for which the driver is being written.

Task Completion Time

The *task completion time* is the time taken for a task to complete execution, including the time to process all interrupts which may occur during the execution of the application task.

NOTE: *The device driver writer should be aware of all delays that the interrupts could potentially cause to an application. This is important when considering the overall responsiveness of the "application-plus-kernel" combination in the worst-possible timing scenario.*

Real Time Response

To improve the real time response of any operating system, the most important parameters are the driver response time, task response time, and the task completion time. The time taken by the driver in the system can have a direct effect on the system's overall real-time response. A single breach of this convention can render a high performance real-time system totally useless (at least in a real-time sense).

Kernel threads were introduced into LynxOS in order to keep drivers from interfering with the real-time response of the overall system. LynxOS kernel threads were designed specifically to increase driver functionality while decreasing driver response time, task response time, and task completion time.

In a normal system, interrupts have a higher priority than any task. A task, regardless of its priority, is interrupted if an interrupt is pending (unless the interrupts have been disabled). The result could mean that a low priority interrupt could interrupt a task which is executing with real-time constraints.

A classic example of this would be a task collecting data for a real-time data acquisition system and being interrupted by a low priority printer interrupt. The task would not continue execution until the interrupt service routine had finished.

With kernel threads, delays of this sort are significantly reduced. Instead of the interrupt service routine doing all the servicing of the interrupt, a kernel thread is used to perform the function previously performed by the interrupt routine. The interrupt service routine is now reduced to merely signalling a semaphore which the kernel thread is waiting on.

Since the kernel thread is running at the application's priority (actually it is running at half a priority level higher as described below), it is scheduled according to process priority and not hardware priority. This ensures that the interrupt service time is kept to a minimum and the task response time is kept short. A further result of this is that the task completion time is also reduced.

The use of kernel threads and priority tracking in LynxOS drivers are the cornerstone to guaranteeing deterministic real-time performance.

Kernel Threads

Kernel threads execute in the virtual memory context of the null process, process 0. However, kernel threads do not have any user code associated with them so context switch times for kernel threads are quicker than for user threads. Like all other tasks in the system, kernel threads have a scheduling priority which the driver can change dynamically to implement priority tracking. They are scheduled with the `SCHED_FIFO` algorithm.

Creating Kernel Threads

A kernel thread is created once in the install or open entry point. The advantage of starting it in the open is that, if device is never opened, the driver doesn't use up kernel resources unnecessarily. However, as the thread is only created once, the open routine must check whether this is the first call to open. One thread is created for each interrupting device, which normally corresponds to a major device. The following code fragment illustrates how a thread might be started from the install entry point:

```
int threadfunc ();
int stacksize, priority;
char *threadname;

s->st_id = ststart (threadfunc, stacksize, priority,
threadname, 1, s);
if (s->st_id == SYSERR) {
    sysfree (s, sizeof (struct statics));
    pseterr (EAGAIN);
    return (SYSERR);
}
```

The thread function specifies a C function which will be executed by the thread. The structure of the thread code is discussed in the next section.

The second argument specifies the thread's stack size. This stack does not grow dynamically, so enough space must be allocated to hold all the thread's local variables.

As kernel threads are pre-emptible tasks, they have a scheduling priority, just like other user threads in the system, which determines the order of execution between tasks. The priorities of kernel threads are discussed more fully in the section on priority tracking. It is usual to create the thread with a priority of 1.

The thread name is an arbitrary character string which is printed in the *name* column by the `ps T` command. It will be truncated to `PNMLEN` characters (including NULL terminator). `PNMLEN` is currently 32, see `<proc.h>`.

The last two parameters allow arguments to be passed to the thread. In most cases, it is sufficient to pass the address of the statics structure, which normally contains all other information the thread might need for communication and synchronization with the rest of the driver.

Structure of a Kernel Thread

The structure of a kernel thread and the way in which it communicates with the rest of the driver depends largely on the way in which a particular device is used. For the purposes of illustration, two different driver designs will be discussed.

1. *Exclusive access*. Only one user task is allowed to use the device at a time. The exclusive access is often enforced in the open entry point.
2. *Multiple access*. Multiple user tasks are permitted to have the device open and make requests.

Exclusive Access

If we consider synchronous transfers only, then this type of driver will typically have the following structure:

1. The top-half entry point (read/write) starts the data transfer on the device, then blocks waiting for I/O completion.
2. The interrupt handler signals the kernel thread when the I/O completes.
3. The kernel thread consists of an indefinite for loop which does the following:
   - Wait for work to do
   - Process interrupt
   - Wake up user task

The statics structure will contain a number of variables for communication between the thread and the other entry points. These would include synchronization semaphores, error status, transfer length etc.

Top-half Entry Point

The read/write entry point code will not be any different from a driver that does not use kernel threads. It starts an operation on the device, then blocks on an event synchronization semaphore.

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    start_IO (s, buff, count, READ);
                          /* start I/O on device */
    swait (&s->io_sem, SEM_SIGABORT);
                         /* wait for I/O completion */
    if (s->error) {       /* check error status */
        pseterr (EIO);
        return (SYSERR);
    }
   return (s->count);   /* return # bytes transfered */
}
```

Interrupt Handler

Apart from any operations that may be necessary to acknowledge the hardware interrupt, the interrupt handler's only responsibility is to signal the kernel thread, informing it that there is some work to do:

```
intr_handler (s)
struct statics *s;
{
   ssignal (&s->intr_sem);  /* wake up kernel thread */
}
```

Kernel Thread

The kernel thread waits on an event synchronization semaphore. When an interrupt occurs, the thread is woken up by the interrupt handler. It processes the interrupt, checking the device status for errors etc. and wakes up the user task that is waiting for I/O completion. For best system real-time performance, the kernel thread should re-enable interrupts from the device.

```
kthread (s)
struct statics *s;
{
    for (;;) {
       swait (&s->intr_sem, SEM_SIGIGNORE);
                                 /* wait for work to do*/
        ...
       /* process interrupt, check for errors etc. */
        ...
        if (error_found)
            s->error = 1;
                /* tell user task there was an error */
       ssignal (&s->io_sem);  /* wake up user task */
    }
}
```

Multiple Access

In this type of design, any number of user tasks can open a device and make requests to the driver. But as most devices can only perform one operation at a time, requests from multiple tasks must be held in a queue. In a system *without* kernel threads, the structure of such a driver is:

1. The top-half routine starts the operation immediately if the device is idle, otherwise it enqueues the request. It then blocks waiting for the request to be completed.
2. The interrupt handler processes interrupts, does all I/O completion, wakes up the user task and then starts the next operation on the device immediately if there are queued requests.

The problem with this strategy is that it can lead to an overly long interrupt routine owing to the large amount of work done in the handler. Since interrupt handlers are not pre-emptible this can have an adverse effect on system response times. When multiple requests are queued up the next operation is started immediately the previous one has finished. The result of this is that a heavily used device can generate a series of interrupts in rapid succession until the request queue is emptied. Even if the requests were made by low priority tasks, the processing of these interrupts and requests will take priority over high priority tasks because it is done within the interrupt handler.

The use of kernel threads resolves these problems by off-loading the interrupt handler. A kernel thread is responsible for dequeuing and starting requests, handling I/O completion and waking up the user tasks. Figure 8-1 illustrates the overall design.

A data structure containing variables for event synchronization, error status etc. is used to describe each request. The pending request queue and list of free request headers will be part of the statics structure. The interrupt handler code is the same as in the case of the exclusive use design.

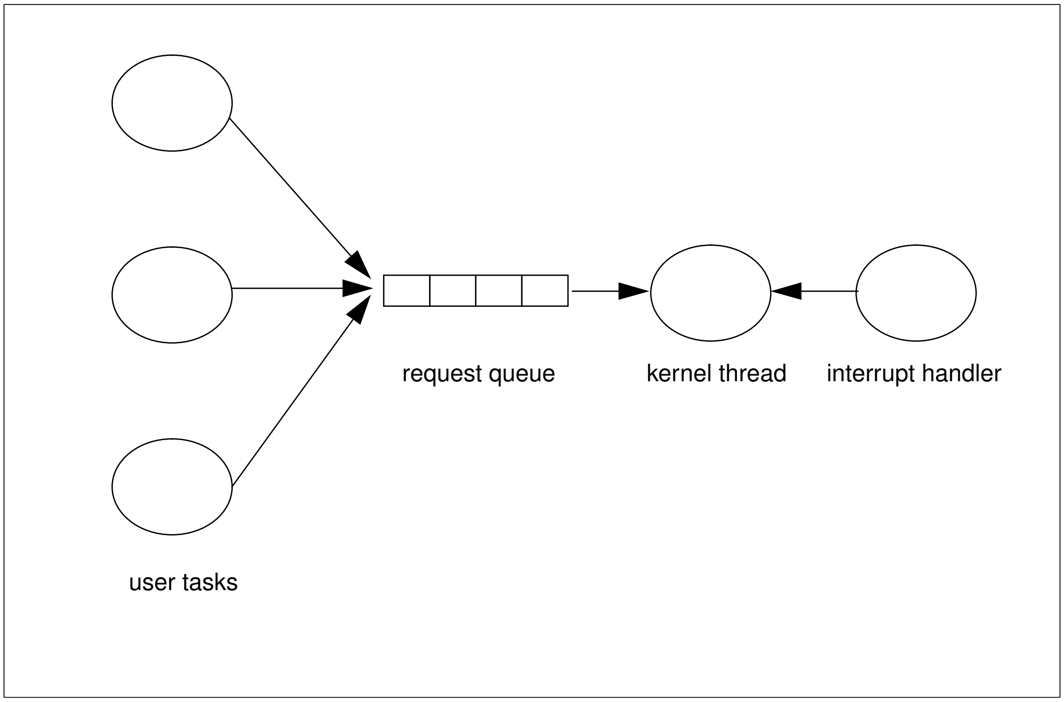

Figure 8-1: Interrupt Handler Design

Top-half Entry Point

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    struct req_hdr *req;

    ...
    enqueue (s, req);        /* enqueue request */
    swait (&req->io_sem, SEM_SIGABORT);
                            /* wait for I/O completion */
    ...
}
```

Kernel Thread

```
kthread (s)
struct statics *s;
{
    struct req_hdr *curr_req;

    for (;;) {
       curr_req = dequeue (s); /* wait for a request */
      start_IO (s, curr_req); /* start I/O operation */
        /* wait for I/O completion */
        swait (&s->intr_sem, SEM_SIGIGNORE);
        ...
        /* process interrupt, check for errors etc. */
        ...
        if (error_found)
            /* tell user task there was an error */
            curr_req->error = 1;
        /* wake up user task */
        ssignal (&curr_req->io_sem);
    }
}
```

Priority Tracking

The previous examples did not discuss the priority of the kernel thread. It was assumed to be set statically when the thread is created. There is a fundamental problem with using a static thread priority in that, whatever priority is chosen, there are always some conceivable situations where the order of task execution does not meet real-time requirements. The same is true of systems that implement separate scheduling classes for system and user level tasks. Figure 8-2 shows two possible scenarios in a system using a static thread priority. In both scenarios, Task A is using a device that generates work for the kernel thread. Other tasks, with different priorities, exist in the system. These are represented by Task B.

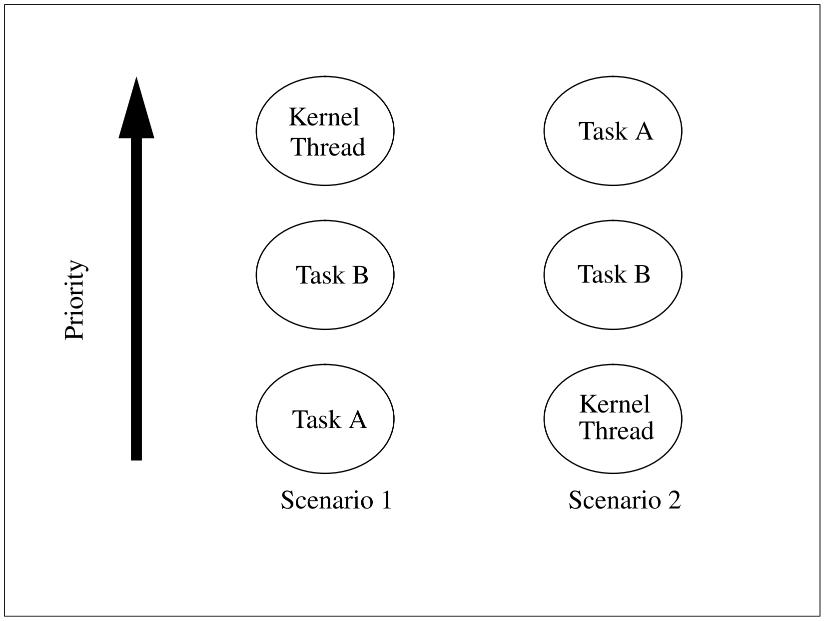

Figure 8-2: Scheduling with Static Thread Priorities

In the first scenario, Task B has a priority higher than Task A but lower than the kernel thread. The kernel thread will be scheduled before Task B even though it is processing requests on behalf of a lower priority task. This is essentially the same situation that occurs when interrupt processing is done in the interrupt handler. In Scenario 2), the situation is reversed. The kernel thread is pre-empted by Task B resulting in Task A being delayed.

The only solution that can meet the requirements of a deterministic real-time system with bounded response times is to use a technique that allows the kernel thread priority to dynamically follow the priorities of the tasks that are using a device.

User and Kernel Priorities

User applications can use 256 priority levels from 0–255. However, internally the kernel uses 512 priority levels, 0–511.

The user priority is converted to the internal representation simply by multiplying it by two, as illustrated in Figure 8-3

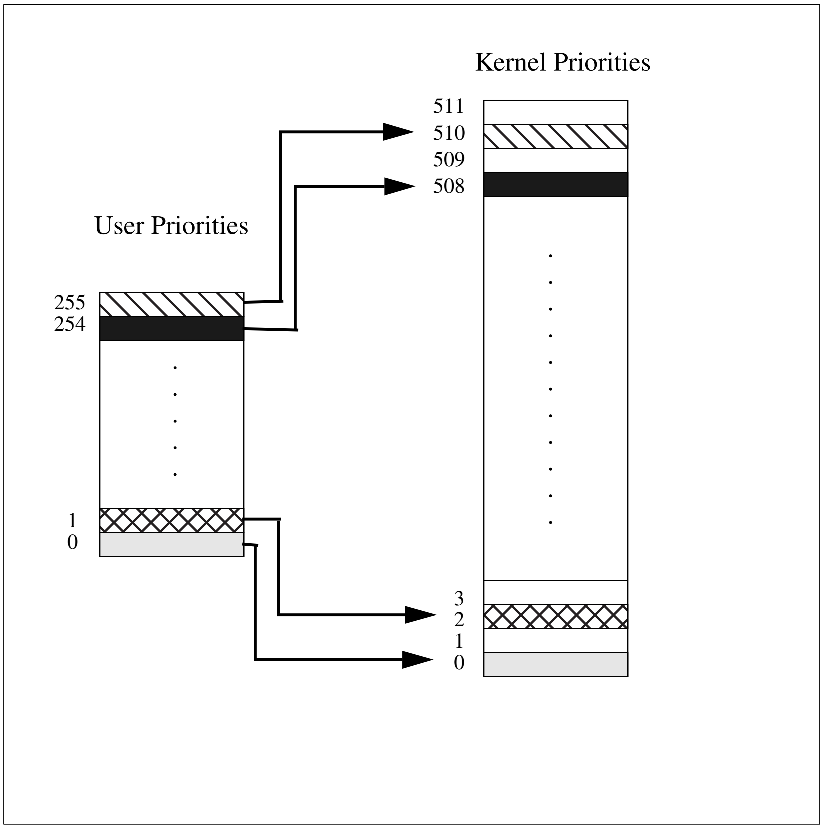

Figure 8-3: User and Kernel Priorities

As can be seen, a user task will always have an even priority at the kernel level. This results in “empty”, odd priority slots between the user priorities. These slots play an important role in priority tracking.

The examples below will again discuss the exclusive and multiple access driver designs for illustrating priority tracking techniques.

Exclusive Access

Whenever a request is made to the driver, the top-half entry point must set the kernel thread priority to the priority of the user task.

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    uprio = _getpriority ();
    /* get priority of current task */
    stsetprio (s->kt_id, (uprio << 1) + 1);
    /* set k.t. priority */
    start_IO (s, buff, count, READ);
    /* start I/O on device */
    swait (&s->io_sem, SEM_SIGABORT);
    /* wait for I/O completion */
    if (s->error) {
     /* check error status */
        pseterr (EIO);
        return (SYSERR);
    }
    return (s->count);
    /* return # bytes transfered */
}
```

The expression (uprio << 1) + 1 converts the user priority to a kernel level priority. The thread priority is in fact set to the odd numbered kernel priority just above the priority of the user task. This ensures that the kernel thread executes before any tasks at the same or lower priority as the user task making the request but after any user tasks of higher priority, as shown in Figure 8-4.

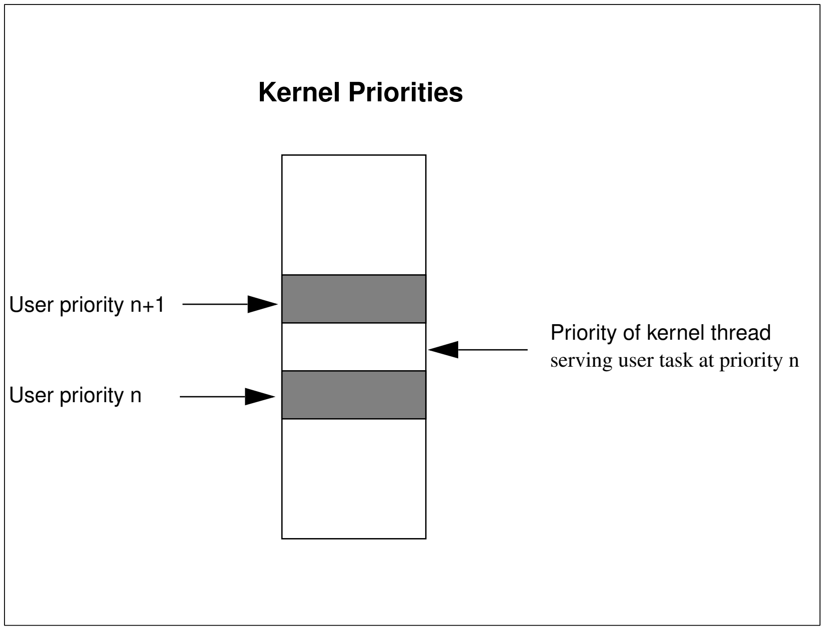

Figure 8-4: Kernel Thread Priorities

When the request has been completed the thread resets its priority to its initial value.

```
kthread (s)
struct statics *s;
{
    for (;;) {
        swait (&s->intr_sem, SEM_SIGIGNORE);
         /* wait for work to do */
         ...
        /* process interrupt, check for errors etc. */
         ...
        if (error_found)
            s->error = 1;
             /*tell user task there was an error*/
        ssignal (&s->io_sem);
         /* wake up user task */
        stsetprio (s->kt_id, 1);
        /* reset kernel thread priority */
    }
}
```

Multiple Access

As was seen in the previous discussion of this type of design, the driver maintains a queue of pending requests from a number of user tasks. These tasks will probably have different priorities so the driver must ensure that the kernel thread is always running at the priority of the highest priority user task which has a request pending. If the requests are queued in priority order this will ensure that the thread is always processing the highest priority request. The thread priority must be checked and adjusted at two places. Whenever a new request is made and whenever a request is completed.

How can the driver keep track of the priorities of all the user tasks that have outstanding requests. In order to do this the driver must use a special data structure, `struct priotrack`, defined in `<st.h>`. Basically, the structure is a set of counters, one counter for each priority level. The value of each counter represents the number of outstanding requests at that priority. The values of the counters are incremented and decremented using the routines `priot_add` and `priot_remove`. The routine `priot_max` returns the highest priority in the set. The use of these routines is illustrated in the following code examples.

Top-half Entry Point

The top-half entry point must first use `priot_add` to add the new request to the set of tracked requests. The code then decides whether the kernel thread's priority must be adjusted. This will be necessary if the priority of the task making the new request is higher than the thread's current priority. A variable in the statics structure is used to track the kernel thread's current priority. The request header must also contain a field specifying the priority of the task making each request. This is used by the kernel thread.

```
drv_read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    ...
   uprio = _getpriority ();  /* get user task priority
*/
    req->prio = uprio;        /* save for later use */
    enqueue (s, req);          /* enqueue request */
    /*
     * Do priority tracking. Add priority of new request
     * to set. If priority of new request is higher than
    * current thread priority, adjust thread priority.
     */
    swait(&s->prio_sem, SEM_SIGIGNORE);
    /* synchronize with kernel thread */
    priot_add (&s->priotrack, uprio, 1);
    if (uprio > s->kt_prio) {
        stsetprio (s->kt_id (uprio << 1) + 1);
        s->kt_prio = uprio;
    }
    ssignal(&s->prio_sem);
    swait (&req->io_sem, SEM_SIGABORT);
    /* wait for I/O completion */
    ...
}
```

Kernel Thread

When the kernel thread has finished processing a request, the priority of the completed request is removed from the set using `priot_remove`. The thread must then determine whether to change its priority or not, depending on the priorities of the remaining pending requests. The thread uses `priot_max` to determine the highest priority pending request.

```
kthread (s)
struct statics *s;
{
    ...
    for (;;) {
        ...
        curr_req = dequeue (s); /* wait for a request */
        start_IO (s, curr_req); /* start I/O operation */
        swait (&s->intr_sem, SEM_SIGIGNORE);
        /* wait for I/O completion */
            ...
        /* process interrupt, check for errors etc. */
            ...
        /*
         * Do priority tracking. Remove priority of
         * completed request from set. Determine high
         * priority of remaining requests. If this is
         * lower than current priority, adjust thread
         * priority.
         */
        swait(&s->prio_sem, SEM_SIGIGNORE);
        /* synchronize with top-half */
        priot_remove (&s->priotrack, curr_req->prio);
        maxprio = priot_max (&s->priotrack);
        if (maxprio < s->kt_prio) {
            stsetprio (s->kt_id, (maxprio << 1) + 1);
            s->kt_prio = maxprio;
        }
        ssignal(&s->prio_sem);
        ...
    }
}
```

Non-atomic Requests

The previous examples implicitly assumed that requests made to the driver are handled atomically, that is to say, the device can handle an arbitrary sized data transfer. This is not always the case. Many devices have a limit on the size of transfer that can be made in which case the driver may have to divide the user data into smaller chunks. A good example is a driver for a serial device. A user task may request a transfer of many bytes, but the device can only transfer one byte at a time. The driver must split the request into multiple single byte requests. From the point of view of priority tracking a single task requesting an n byte transfer is equivalent to n tasks requesting single byte transfers. Since each byte is handled as a separate transfer by the driver (each byte generates an interrupt), the priority tracking counters must count the number of bytes rather than the number of requests.

The functions `priot_add`*n* and `priot_remove`*n* can be used to add and remove multiple requests to the set of tracked priorities. What is defined as a request depends on the way the driver is implemented. It will not always correspond on a one to one basis with a request at the application level. Taking again the example of a driver for a serial device, a single request at the application level consists of a call to the driver to transfer a buffer of length n bytes. However, the driver will split the buffer into n single byte transfers, each byte representing a request at the driver level. The top-half entry point would add n requests to the set of tracked priorities using `priot_add`*n*. As each byte is transferred, the kernel thread would remove each request priority using `priot_remove`. The priority of the kernel thread would only be updated when all bytes have been transferred. It is very important that the priority tracking is based on requests as defined at the driver level, not the application level, in order for the priority tracking to work correctly.

Controlling Interrupts

One of the problems that was discussed concerning drivers that perform all interrupt processing in the interrupt handler was the fact that in certain circumstances a device can generate a series of interrupts in rapid succession. For many devices, the use of the kernel thread and priority tracking techniques illustrated above resolves the problem.

One example concerns a disk driver. Figure 8-5 represents a situation that can occur in a system without kernel threads. A lower priority task makes multiple requests to the driver. Before these requests have completed, a higher priority task starts execution. But this higher priority task is continually interrupted by the interrupt handler for the disk. Because of the amount of processing that can be done within the interrupt handler and because the number of requests that could have been queued up for the disk is unlimited, the response time of the system and execution time for the higher priority task is essentially unbounded.

Figure 8-6 shows the same scenario using kernel threads. The important thing to note is that the higher priority task can only be interrupted once by the disk. The kernel thread is responsible for starting the next operation on the disk but because the kernel thread's priority is based on task B's priority, it will not run until the higher priority task has completed. In addition the length of time during which task A is interrupted by the interrupt handler is a small constant time as the majority of the interrupt processing has been moved to the kernel thread.

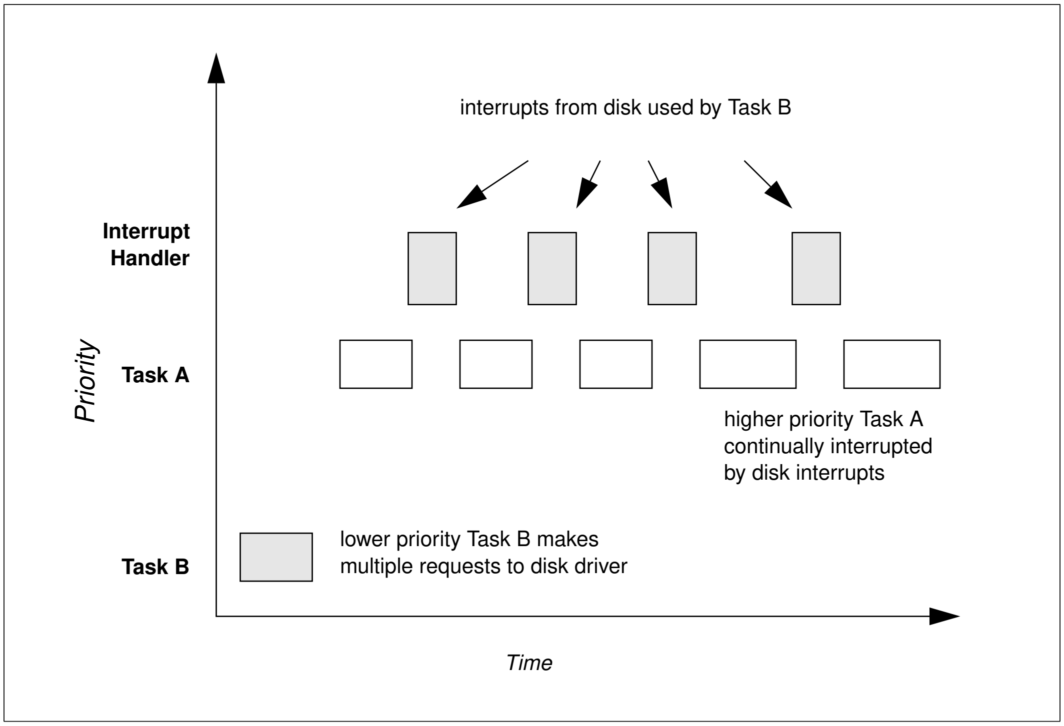

Figure 8-5: Interrupt Handling without Kernel Threads

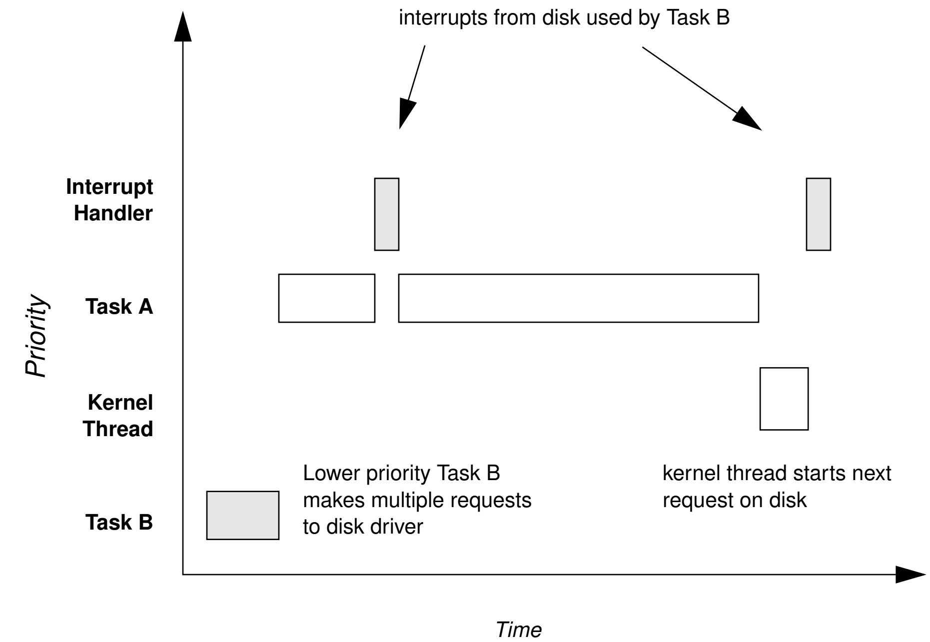

Figure 8-6: Interrupt Handling with Kernel Threads

This scheme takes care of devices where requests are generated by local tasks. But what about devices where data is being sent from a remote system? The local operating system cannot control when or how many packets are received over an Ethernet connection, for example. Or a user typing at a keyboard could generate multiple interrupts. The solution to these situations is again based on the use of kernel threads. For such devices, the interrupt handler must disable further interrupts from the device. Interrupts are then re-enabled by the corresponding kernel thread. So again, a device can only generate a single interrupt until the thread has been scheduled to run. Any higher priority tasks will execute to completion before the thread and can be interrupted by a maximum of one interrupt from each device. The use of this technique requires that the device has the ability to store some small amount of incoming data locally during the time that its interrupt is disabled. This is not usually a problem for most devices.

Example

In the following example, we show how the previous printer device driver can be implemented with kernel threads:

```
/* ptrinfo.h  */
struct ptrinfo {
    int port;
    int irq;
    int qlen;
};

/* ptrinfo.c */
#include "ptrinfo.h"

struct ptrinfo ptrinfo0 = {
    /* port */ 0x378,
    /* irq  */ 7,
    /* qlen */ 100
};


/* ptrioctl.h  */
#define PTRSTATUS 500

struct ptrstatus {
    int chars;             /* characters printed */
int lines;             /* lines printed      */
};

/* ptrdrvr.c - using threads */
#include <kernel.h>
#include <mem.h>
#include <file.h>
#include <errno.h>
#include <ioctl.h>
#include <conf.h>
#include <st.h>
#include "ptrinfo.h"
#include "ptrioctl.h"
```

```
/* ports  */
#define PP_DATA    0         /* data port offset       */
#define PP_STATUS  1         /* status port offset     */
#define PP_CONTROL 2         /* control port offset    */

/* status bits         */
#define PP_BUSY    0x80      /* printer busy           */
#define PP_PE      0x20      /* out of paper           */
#define PP_SLCT    0x10      /* printer is selected    */
#define PP_ERROR   0x08      /* printer detected error */

/* control bits        */
#define PP_IENABLE 0x10
/* interrupt enable    */
#define PP_SLCTIN  0x08      /* select printer  */
#define PP_INIT    0x04      /* start printer   */
#define PP_AUTOLF  0x02      /* auto line feed  */
#define PP_STROBE  0x01      /* strobe printer  */

typedef unsigned short ptype;

#define STACKSIZE PAGESIZE

struct qentry {
    char c;                  /* character    */
    int pri;                 /* its priority */
};

struct ptrstatics {
    ptype datap;             /* data port address   */
    ptype controlp;          /* cntrl port address  */
    char control;            /* control bits        */
    int irq;                 /* IRQ number          */
    int closing;             /* closing device      */
    int close_sem;           /* sempahore for close */
    int expecting;           /* expecting an int.?  */
    int nextnl;              /* output a '\r' next? */
    int chars;               /* printed since open  */
    int lines;               /* printed since open  */
    int qlen;                /* characters in queue */
    struct qentry *q;        /* the queue itself    */
    int head;                /* head of queue       */
    int tail;                /* tail of queue       */
    int qdata;               /* data in the queue   */
    int free_sem;            /* free queue space    */
```

```
    int stid;                /* thread id            */
    int int_sem;             /* interrupt semaphore */
    int qsem;                /* queue protection    */
    struct priotrack pt;     /* pri. tracking       */
    int curpri;              /* current priority    */
    int prio_sem;            /* pri. trking sem     */
};

static port_in(), port_out();
asm {
    port_in:                 /* byte = port_in(port) */
        mov EAX, 0
        mov EDX, 4[ESP]
        in  AL, DX
        ret
    port_out:                /* port_out(byte, port) */
        mov EDX, 8[ESP]
        mov EAX, 4[ESP]
        out DX, AL
        ret
}

#define PERR (char *) SYSERR

char *ptrinstall(info)
struct ptrinfo *info;
{
    struct ptrstatics *s;
    static void ptrint(), ptrthread(), ptruninstall();
    int i;
    /* probe for the printer */
    port_out(1, info->port+PP_DATA);
    if (port_in(info->port+PP_DATA) != 1)
        return PERR;
    s = (struct ptrstatics *)
            sysbrk((long)sizeof *s);
    if (!s) return PERR;
    s->q = (struct qentry *)
            sysbrk((long)info->qlen *
                    sizeof(struct qentry));
    if (!s->q) {
        sysfree(s, (long)sizeof *s);
        return PERR;
    }
    /* initialize statics */
    s->datap = info->port + PP_DATA;
```

```
    s->controlp = info->port + PP_CONTROL;
    s->control = PP_SLCTIN | PP_INIT;
    s->irq = info->irq;
    s->expecting = 0;
    s->lines = s->chars = 0;
    s->closing = s->close_sem = 0;
    s->nextnl = 0;
    s->free_sem = s->qlen = info->qlen;
    s->qdata = s->head = s->tail = 0;
    s->int_sem = 0;
    s->qsem = -1;
    bzero(&s->pt, sizeof(struct priotrack));
    s->curpri = 0;
    s->prio_sem = -1;

    /* initialize printer */
    iointset(32+s->irq, ptrint, s);
    port_out(PP_SLCTIN, s->controlp);
    for (i = 0; i < 100; i++) ;
    port_out(s->control, s->controlp);

    s->stid = ststart(ptrthread, STACKSIZE,
            s->curpri, "ptr thread", 1, s);
    if (s->stid == SYSERR) {
        ptruninstall(s);
        return PERR;
    }
    return (char *) s;
}
void ptruninstall(s)
struct ptrstatics *s;
{
    if (s->stid != SYSERR) stremove(s->stid);
    iointclr(32+s->irq);
    sysfree(s->q, (long)s->qlen *
            sizeof(struct qentry));
    sysfree(s, (long)sizeof *s);
}
int ptropen(s, devno, f)
struct ptrstatics *s;
int devno;
struct file *f;
{
    if (f->access_mode & FREAD) {
        pseterr(EINVAL);
        return SYSERR;
```

```
    }
    if (minor(devno)) {
        pseterr(ENXIO);
        return SYSERR;
    }
    return OK;
}
int ptrclose(s, f)
struct ptrstatics *s;
struct file *f;
{
    swait(&s->qsem, SEM_SIGIGNORE);
    if (s->expecting) {
        s->closing = 1;
        ssignal(&s->qsem);
        swait(&s->close_sem, SEM_SIGIGNORE);
        s->closing = 0;
    } else {
        ssignal(&s->qsem);
    }
    s->lines = s->chars = 0;
    return OK;
}
int ptrselect()
{
    return OK;
}
    /*  assumes:
    **    data in the queue
    **    queue access disabled
    */
void send(s)
struct ptrstatics *s;
{
    int prio;
    char c;
    if (s->nextnl) {
        c = '\r';
        prio = s->nextnl;
        s->nextnl = 0;
        s->lines++;
    } else {
        c = s->q[s->head].c;
        prio = s->q[s->head++].pri;
        s->head %= s->qlen;
        s->qdata--;
```

```
            ssignal(&s->free_sem);
            s->nextnl = c == '\n'? prio : 0;
            s->chars++;
        }
        port_out(c, s->datap);
        port_out(s->control | PP_STROBE, s->controlp);
        port_out(s->control | PP_IENABLE, s->controlp);
        if (!s->nextnl) {
            swait(&s->prio_sem, SEM_SIGIGNORE);
            priot_remove(&s->pt, prio);
            if (prio == s->curpri) {
                prio = priot_max(&s->pt);
                if (prio != s->curpri) {
                    s->curpri = prio;
                    stsetprio(s->stid, (prio<<1)+1);
                }
            }
            ssignal(&s->prio_sem);
        }
}
void ptrint(s)
struct ptrstatics *s;
{
    ssignal(&s->int_sem);
}
void ptrthread(s)
struct ptrstatics *s;
{
    for (;;) {
        swait(&s->int_sem, SEM_SIGIGNORE);
        swait(&s->qsem, SEM_SIGIGNORE);
        if (s->qdata || s->nextnl) {
            send(s);
        } else {
                s->expecting = 0;
                /* disable ptr interrupts: */
                port_out(s->control, s->controlp);
                if (s->closing) ssignal(&s->close_sem);
            }
        ssignal(&s->qsem);
    }
}
```

This function `ptrthread()` is really the only difference between this driver and the others. It is a kernel thread used by the driver to send characters out to the printer.

```
int ptrwrite(s, f, buff, count)
struct ptrstatics *s;
struct file *f;
char *buff;
int count;
{
    int i = count, myprio;

    myprio = _getpriority();
    swait(&s->prio_sem, SEM_SIGIGNORE);
    priot_addn(&s->pt, myprio, count);
    if (myprio > s->curpri) {
        s->curpri = myprio;
        stsetprio(s->stid, (myprio<<1)+1);
    }
    ssignal(&s->prio_sem);

    while (i--) {
        while (swait(&s->free_sem, SEM_SIGABORT)) {
            swait(&s->prio_sem, SEM_SIGABORT);
            priot_removen(&s->pt, myprio, i+1);
            if (s->curpri == myprio) {
                s->curpri = priot_max(&s->pt);
                if (s->curpri < myprio) {
                    stsetprio(s->stid,(s->curpri<<1)+1);
                }
            }
            ssignal(&s->prio_sem);
            deliversigs();

            priot_addn(&s->pt, myprio, i+1);
            if (myprio > s->curpri) {
                s->curpri = myprio;
                stsetprio(s->stid, (myprio<<1)+1);
            }
            ssignal(&s->prio_sem);
        }
        swait(&s->qsem, SEM_SIGIGNORE);
        s->q[s->tail].c = *buff++;
        s->q[s->tail++].pri = myprio;
        s->tail %= s->qlen;
        s->qdata++;
        if (!s->expecting) {
            send(s);
            s->expecting = 1;
        }
```

```
        ssignal(&s->qsem);
    }
    return count;
}
int ptrioctl(s, f, command, arg)
struct ptrstatics *s;
struct file *f;
int command;
char *arg;
{
    switch (command) {
        case PTRSTATUS:
           if (wbounds(arg) < sizeof(struct ptrstatus)) {
                pseterr(EFAULT);
                return SYSERR;
            }
            ((struct ptrstatus *)arg)->chars = s->chars;
            ((struct ptrstatus *)arg)->lines = s->lines;
            break;
        case FIOPRIO:
        case FIOASYNC:
            break;
        default:
            pseterr(EINVAL);
            return SYSERR;
    }
    return OK;
}
#include <dldd.h>
static struct dldd entry_points = {
    ptropen, ptrclose, 0, ptrwrite,
    ptrselect, ptrioctl,
    ptrinstall, ptruninstall, (char *) 0
};
```

CHAPTER 9

Network Device Drivers

A network driver is defined as a link-level device driver that interfaces to the LynxOS TCP/IP module. Unlike other drivers, a network driver does not interface directly with user applications. It interfaces instead with the LynxOS TCP/IP module. This interface is defined by a set of driver entry points and data structures, described in the following sections.

Kernel threads play an important role in LynxOS networking software, not only within the drivers, but also as part of the TCP/IP module.

The example code fragments given below as illustration of the points discussed are for an Ethernet device. These examples can easily be adapted to other technologies.

Kernel Data Structures

A network driver must make use of a number of kernel data structures. These are each briefly described here and their use is further illustrated throughout the rest of the chapter.

A driver must include the following header files which define these structures and various symbolic constants that are used in the rest of this chapter:

```
#include <types.h>
#include <io.h>
#include <ioctl.h>
#include <socket.h>
#include <bsd/in.h>
#include <bsd/if.h>
#include <bsd/if_ether.h>
#include <bsd/in_var.h
#include <bsd/bsd_mbuf.h>
#include <bsd/netisr.h>
```

struct ether_header

The Ethernet header must be prefixed to every outgoing packet. It specifies destination and source Ethernet addresses and a packet type. The symbolic constants `ETHERTYPE_IP`, `ETHERTYPE_ARP` and `ETHERTYPE_RARP` can be used for the packet type.

```
struct ether_header {
   u_char  ether_dhost[6];   /* dest Ethernet addr */
   u_char  ether_shost[6]; /* source Ethernet addr */
   u_short ether_type;     /* Ethernet packet type */
}
```

struct arpcom

The `arpcom` structure is used for communication between the TCP/IP module and the network interface driver. It contains the `ifnet` structure (described below) and the interface's Ethernet and Internet addresses. This structure must be the first element in the statics structure.

```
struct arpcom {
    struct ifnet ac_if;/* network visible interface */
    u_char ac_enaddr[6];        /* Ethernet address */
    struct in_addr ac_ipaddr;   /* Internet address */
    struct ether_multi *ac_multiaddrs;
```

```
                       /* list of ether multicast addrs */
    int ac_multicnt;/* length of ac_multiaddrs list */
};
```

struct sockaddr

A generic structure for specifying socket addresses, containing an address family field and up to 14 bytes of protocol-specific address data.

```
struct sockaddr {
    u_char sa_len;                    /* total length */
    u_char  sa_family;              /* address family */
    char sa_data[14];           /* longer; addr value */
};
```

struct sockaddr_in

A structure used for specifying socket addresses for the Internet protocol family.

```
struct sockaddr_in {
    u_char sin_len;
    u_char sin_family;              /* always AF_INET */
    u_short sin_port;                  /* port number */
    struct in_addr sin_addr; /* host Internet addr */
    char sin_zero[8];
};
```

struct in_addr

Structure specifying a 32-bit host Internet address.

```
struct in_addr {
    u_long s_addr;
};
```

struct ifnet

This is the principle data structure used to communicate between the driver and the TCP/IP module. It provides the TCP/IP software with a generic, hardware independent interface to the network drivers. It specifies a number of entry points that the TCP/IP module can call in the driver, a flag variable indicating general characteristics and the current state of the interface, a queue for outgoing packets and a number of statistics counters.

```
struct ifnet {
    char *if_name;       /* name, e.g. "wd" or "oblan" */
    char *p;                     /* user defined field */
    struct ifnet *if_next;
                      /* all struct ifnets are chained */
    struct ifaddr *if_addrlist;
                           /* linked list of addresses */
    int if_pcount;  /* number of promiscuous listeners */
    caddr_t if_bpf;         /* packet filter structure */
    u_short if_index;   /* numeric abbreviation for if */
    short if_unit;  /* sub-unit for lower level driver */
    short if_timer;   /* time 'til if_-watchdog called */
    short if_flags;        /* up/down, broadcast, etc. */

    struct if_data {
    /* generic interface information */
        u_char ifi_type;    /* Ethernet, tokenring etc */
        u_char ifi_addrlen;    /* media address length */
        u_char ifi_hdrlen;      /* media header length */
        u_long ifi_mtu;   /* maximum transmission unit */
        u_long ifi_metric;/* routing metric (external) */
        u_long ifi_baudrate;             /* line speed */

    /* volatile statistics */
        u_long ifi_ipackets;/* packets received on i/f */
        u_long ifi_ierrors;     /* input errors on i/f */
        u_long ifi_opackets;    /* packets sent on i/f */
        u_long ifi_oerrors;     /* ouput errors on i/f */
        u_long ifi_collisions;
                             /* collisions on csma i/f */
        u_long ifi_ibytes;
                      /* total number of bytes received */
        u_long ifi_obytes;
                         /* total number of octets sent */
        u_long ifi_imcasts;
                      /* packets received via multicast */
        u_long  ifi_omcasts;
                          /* packets sent via multicast */
        u_long  ifi_iqdrops;
                    /* dropped on input, this interface */
        u_long  ifi_noproto;
```

```
                    /* destined for unsupported protocol */
        struct  timeval ifi_lastchange;/* last updated */
    } if_data;

/* procedure handles */
    int (*if_init)();                      /* init routine */
    int (*if_output)();                  /* output routine */
    int (*if_start)();         /* initiate output routine */
    int (*if_done)();          /* output complete routine */
    int (*if_ioctl)();                    /* ioctl routine */
    int (*if_reset)();                /* bus reset routine */
    int (*if_watchdog)();                 /* timer routine */
    int (*if_setprio)();   /* prio tracking of kthread */

/* output queue */
    struct ifqueue {
        struct mbuf *ifq_head;
        struct mbuf *ifq_tail;
        int  ifq_len;
        int  ifq_maxlen;
        int  ifq_drops;
    } if_snd;
    struct raweth *if_raweth;
};
```

The symbolic constants `IFF_UP`, `IFF_RUNNING`, `IFF_BROADCAST` and `IFF_NOTRAILERS` can be used to set bits in the `if_flags` field.

Looking at the `arpcom` structure, you will notice that the first member is an `ifnet` structure. A driver should declare a struct `arpcom` as part of the statics structure and use the `ifnet` structure within this. There is an important reason for this, explained below in the discussion of the `ioctl` entry point.

struct mbuf

Data packets are passed between the TCP/IP module and a network interface driver using `mbuf` structures. This structure is designed to allow the efficient encapsulation and decapsulation of protocol packets without the need for copying data. A number of functions and macros are defined in `<mbuf.h>` for using `mbuf` structures.

```
/* header at beginning of each mbuf: */
struct m_hdr {
    struct  mbuf *mh_next;     /* next buffer in chain */
    struct  mbuf *mh_nextpkt;   /* next chain in queue */
    int mh_len;         /* amount of data in this mbuf */
    caddr_t mh_data;               /* location of data */
    short   mh_type;      /* type of data in this mbuf */
    short   mh_flags;              /* flags; see below */
};

/* record/packet header in first mbuf of chain;
 * valid if M_PKTHDR set
 */

struct  pkthdr {
    int len;                     /* total packet length */
    struct  ifnet *rcvif;              /* rcv interface */
};

/* description of external storage mapped into mbuf,
 * valid if M_EXT set
 */

struct m_ext {
    caddr_t ext_buf;                 /* start of buffer */
    void (*ext_free)();                 /* free routine */
    u_int   ext_size;  /* size of buffer, for ext_free */
};

struct mbuf {
    struct  m_hdr m_hdr;
    union {
        struct {
            struct  pkthdr MH_pkthdr;  /* M_PKTHDR set */
            union {
                struct  m_ext MH_ext;     /* M_EXT set */
                char    MH_databuf[MHLEN];
            } MH_dat;
        } MH;
        char  M_databuf[MLEN];    /* !M_PKTHDR, !M_EXT */
    } M_dat;
};

#define m_next      m_hdr.mh_next
#define m_len       m_hdr.mh_len
#define m_data      m_hdr.mh_data
```

```
#define m_type      m_hdr.mh_type
#define m_flags     m_hdr.mh_flags
#define m_nextpkt   m_hdr.mh_nextpkt
#define m_act       m_nextpkt
#define m_pkthdr    M_dat.MH.MH_pkthdr
#define m_ext       M_dat.MH.MH_dat.MH_ext
#define m_pktdat    M_dat.MH.MH_dat.MH_databuf
#define m_dat       M_dat.M_databuf
```

Adding/Removing Data in a mbuf

The position and number of data currently in an mbuf are identified by a pointer and a length. By changing these values, data can be added or deleted at the beginning or end of the mbuf. A pointer to the start of the data in the mbuf can be obtained using the mtod macro. The pointer is cast to an arbitrary data type, specified as an argument to the macro:

```
char *cp;
struct mbuf *mb;

cp = mtod (mb, char *);
/* get pointer to data in mbuf */
```

The macro dtom takes a pointer to data anywhere within the data portion of the mbuf and returns a pointer to the mbuf structure itself. For example, if we know that cp points within the data area of an mbuf, the sequence:

```
struct mbuf *mb;
char *cp;

mb = dtom(cp);
```

Data is added to the head of an mbuf by decrementing the m_data pointer, incrementing the m_len value and copying the data using a function such as bcopy. Data is added to the tail of an mbuf in a similar manner by incrementing the m_len value. The ability to add data to the tail of an mbuf is useful for implementing trailer protocols but LynxOS does not currently support such protocols.

Data is removed from the head or tail of an mbuf by simply incrementing the m_data pointer and/or decrementing m_len.

Allocating mbufs

The above examples did not discuss what to do when sufficient space is not available in an mbuf to add data. In this case, a new mbuf can be allocated using the function m_get. The new mbuf is linked onto the existing mbuf chain using its m_next field. m_get can be replaced with MGET which is a macro rather than a function call. MGET produces faster code whereas m_get results in smaller code. The example to add data to the beginning of a packet now becomes:

```
struct mbuf *m;
caddr_t src, dst;

MGET(m, M_DONTWAIT, MT_HEADER);
if (m == NULL)
    return (ENOBUFS);
dst = mtod (m, caddr_t);
bcopy (src, dst, n);
```

The second argument to m_get/MGET specifies whether the function should block or return an error when no mbufs are available. A driver should use M_DONTWAIT which will cause the mbuf pointer to be set to zero if no mbufs are free.

The third argument to m_get/MGET specifies how the mbuf will be used. This is for statistical purposes only and is used, for example, by the command netstat -m. The types used by a network driver are MT_HEADER for protocol headers and MT_DATA for data packets.

mbuf Clusters

When receiving packets from a network interface, a driver must allocate mbufs to store the data. If the data packets are large enough, a structure known as an mbuf cluster can be used. A cluster can hold more data than a regular mbuf, MCLBYTES bytes as opposed to MLEN. As a rule of thumb, there is benefit to be gained from using a cluster if the packet is larger than MCLBYTES/2.

Freeing mbufs

Since there are a limited number of mbufs in the system, the driver must take care to free mbufs at the appropriate points, listed below:

Packet Output:

- Interface is down.
- Address family not supported
- No mbufs for Ethernet header
- if_snd queue is full
- After packet has been transferred to interface

Packet Input:

- Not enough mbufs to receive packet
- Unknown Ethernet packet type
- Input queue is full

The sections on packet input and output show appropriate code examples for each of the above situations. Failure to free them will eventually lead to the system running out of mbufs.

Table 9-1: Summary of Commonly Used mbuf Macros

Macro	Description
MCLGET	Get a cluster and set the data pointer of the mbuf to point to the cluster.
MFREE	Free the mbuf. On return mbuf's successor (pointed to by m->m_next) is stored in the second argument.
MGETHDR	Allocate an mbuf and initialize it as a packet header.
MH_ALIGN	Set the m_data pointer to an mbuf containing a packet header to place an object of the specified size at the end of mbuf, longword aligned.

Table 9-1: Summary of Commonly Used mbuf Macros

Macro	Description
M_PREPEND	Prepend specified bytes of data in front of the data in the mbuf.
dtom	Convert the data pointer within mbuf to mbuf pointer
mtod	convert mbuf pointer to data pointer of specified type.

Table 9-2: Summary of Commonly Used mbuf Functions

Function	Description
m_adj	Remove data from mbuf at start /end.
m_cat	Concatenate one mbuf chain to another.
m_copy	Version of m_copym which does not wait.
m_copydata	Copy data from mbuf chain to a buffer.
m_copyback	Copy data from buffer to a mbuf chain.
m_copym	Create a new mbuf chain from an existing mbuf chain.
m_devget	Create a new mbuf chain with a packet header from data in a buffer.
m_free	A function version of MFREE macro.
m_freem	Free all mbufs in a chain
m_get	A functional version of MGET macro.
m_getclr	Get a mbuf and clear the buffer.
m_gethdr	A function version of the MGETHDR macro.
m_pullup	Pullup data so that the certain number of bytes of data are stored contiguously in the first mbuf in the chain.

Statics Structure

In keeping with the general design philosophy of LynxOS drivers, network drivers should define a statics structure for all device specific information. However, the TCP/IP software has no knowledge of this structure, which is specific to each network interface, and does not pass it as an argument to the driver entry points. The solution to this problem is for the `ifnet` structure to be contained within the statics structure. The user defined field, p, in the `ifnet` structure is initialized to contain the address of the statics structure.

Given the address of the `ifnet` structure passed to the entry point from the TCP/IP software, the driver can obtain the address of the statics structure as follows:

```
struct ifnet *ifp;
struct statics *s = (struct statics *) ifp->p;
```

The `arpcom` structure *must* be the first element in the statics structure. In the code examples below, the `arpcom` structure is named `ac`.

Packet Queues

A number of queues are used for transferring data between the interface and the TCP/IP software. There is an output queue for each interface, contained in the `ifnet` structure. There are two input queues used by all network interfaces. One for IP packets and another for ARP/RARP packets. All queues are accessed using the macros `IF_ENQUEUE`, `IF_DEQUEUEm` `IF_QFULL` and `IF_DROP`.

Driver Entry Points

A network driver contains the following entry points:

install/uninstall	Called by the kernel in the usual manner. The `install` routine must perform a number of tasks specific to network drivers.
interrupt handler	The interrupt handler is declared and called in exactly the same manner as for other drivers. It will not be discussed further here.
output	Called by TCP/IP software to transmit packets on the network interface
ioctl	Called by TCP/IP software to perform a number of commands on the network interface.
watchdog	Called by the TCP/IP software after a user specified timeout period.
reset	Called by the kernel during the reboot sequence.
setprio	Called by the TCP/IP software to implement priority tracking.

By convention, the entry point names are prefixed with the driver name.

Install Entry Point

In addition the usual things done in the `install` routine (allocation and initialization of the statics structure, declaration of interrupt handler etc.), the driver must also fill in the fields of the `ifnet` structure and make the interface known to the TCP/IP software. Note also that hardware initialization is normally done in the `ioctl` routine rather than in the `install` routine.

Finding the Interface Name

The `install` routine must initialize the `if_name` field in the `ifnet` structure. This is the name by which the interface is known to the TCP/IP software. It is used, for example, as an argument to the `ifconfig` and `netstat` utilities.

The interface name is a user defined field specified in the driver's device configuration file (`drvr.cfg`) in the `/sys/lynx.os` directory. The usual technique used by the driver to find this field is to search the `ucdevsw` table for an entry with a matching device information structure address. `ucdevsw` is a kernel table containing entries for all the character devices declared in the `CONFIG.TBL` file. The kernel variable `nucdevsw` gives the size of this table.

```
extern int nucdevsw;
extern struct udevsw_entry ucdevsw[];
struct statics *s;
struct ifnet *ifp;

ifp->if_name = (char *) 0;
for (i = 0; i < nucdevsw; i++) {
    if (ucdevsw[i].info == (char *) info) {
        if (strlen (ucdevsw[i].name) > IFNAMSIZ) {
            sysfree (s, (long) sizeof (struct statics));
            return ((char *) SYSERR);
        }
        ifp->if_name = ucdevsw[i].name;
        break;
    }
}
if (ifp->if_name == (char*) 0) {
    sysfree (s, (long) sizeof (struct statics));
    return ((char *) SYSERR);
}
```

Note that this method only works for statically installed drivers. Dynamically installed drivers do not have an entry in the `ucdevsw` table.

Initializing the TCP/IP Software

A network driver's `install` routine must also test whether the TCP/IP software has been initialized. If it hasn't, it must call the `bsd_sysinit` function:

```
extern int bsdinit_tobedone;

if (bsdinit_tobedone)
    bsd_sysinit();
```

This must be done *before* calling the `if_attach` function.

Initializing the Ethernet Address

The `ac_enaddr` field in the `arpcom` structure is used to hold the interface's Ethernet address, which must be included in the Ethernet header added to all outgoing packets. The `install` routine should initialize this field by reading the Ethernet address from the hardware.

```
struct statics *s;

get_ether_addr (&s->ac.ac_enaddr);
```

In the above example, `get_ether_addr` would be a user written function that reads the Ethernet address from the hardware.

Initializing the ifnet Structure

The various fields in the `ifnet` structure should be initialized to appropriate values. Unused fields should be set to zero or `NULL`. Once the structure has been initialized the network interface is made known to the TCP/IP module by calling the `if_attach` function.

```
struct statics *s;
struct ifnet *ifp;

ifp->if_timer = 0 ;
ifp->p = (char *) s;/* address of statics structure */
ifp->if_unit = 0;
ifp->if_mtu = ETHERMTU;
ifp->if_flags = IFF_BROADCAST | IFF_NOTRAILERS;
```

```
ifp->if_init = NULL;
ifp->if_output = ether_output;
ifp->if_ioctl = drvr_ioctl;
ifp->if_reset = drvr_reset;
ifp->if_start = drvr_start;
ifp->if_setprio = drvr_setprio;
ifp->if_watchdog = drvr_watchdog;

if_attach (ifp);
```

Note that the `if_output` handle in the ifnet structure should point to `ether_output` routine in the TCP/IP module. Previously it pointed to driver specific local routine. In BSD 4.4 most of hardware independent output code has been moved to `ether_output` routine. After the `ether_output` has packaged the data for output, it calls a start routine specified by `if_start` member of interface `ifnet` structure. For example:

```
ifp->if_start = lanstart;
```

Packet Output

The processing of outgoing packets is divided into two parts. The first part concerns the TCP/IP module which is responsible for queueing the packet on the interface's output queue. The actual transmission of packets to the hardware is handled by the driver start routine and the kernel thread. The driver is responsible in all cases for freeing the mbufs holding the data once the packet has been transmitted or at any time an error is encountered.

ether_output Function

A number of tasks previously performed by driver output routine are now done in TCP/IP module by `ether_output` routine. Thus the `if_output` field of interface ifnet structure is initialized to address of `ether_output` routine in the driver install routine:

```
ifp->if_output = ether_output;
```

This causes ether_output routine to be called indirectly when TCP/IP module has a packet to transmit. After enqueuing the packet to transmit, ether_output calls a device-specific function indirectly through if_start pointer in the ifnet structure. For example if ifp points to an ifnet structure,

```
(*ifp->if_start)(ifp)
```

if_start field is also initialized by the driver install routine. The driver start routine starts output on the interface if resources are available. Before removing packets from the output queue for transmission, the code will normally have to test whether the transmitter is idle and ready to accept a new packet. It will typically dequeue a packet (which is enqueued by ether_output) and transmit it.

```
    sdisable(s);
    if (ds->xmt_pending) {
/* if already one transmission is in progress */
        srestore(s);
        return 1;
    }
    IF_DEQUEUE(&ifp->if_snd, m);
    if (m == 0) {
        srestore(s);
        return 0;
    }
    ds->xmt_pending = 1;
    srestore(s);

    ...
    ...
    /* Initiate transmission */
    return 0;
```

Note that manipulation of the output queue must be synchronized with sdisable/srestore. Once the packet has been transferred, the driver must free the mbuf chain using m_freem.

One important point to consider here is that the start routine can now be called by TCP/IP module (via ether_output) and driver stream task upon receiving an interrupt. Thus the start routine must protect code/data in critical area. For example, it could check a “pending” flag which is set before starting to transmit and cleared when transmit done interrupt is received. If the transmit start routine is not re-entrant, it could signal a semaphore in order to notify the driver’s kernel thread that packets are now available on the output queue. The routine should then return 0 to indicate success.

```
ssignal(&s->thread_sem);
return (0);
```

Also note that total data available in an `mbuf` can be obtained from `mbuf` packet header. This is a new feature in BSD 4.4. Previously one had to traverse the `mbuf` chain to calculate it.

```
/* put mbuf data into TFD data area */
length = m->m_pkthdr.len;
m_copydata(m, 0, length, p);
m_freem(m);
```

Kernel Thread Processing

The kernel thread must perform the following activities relating to packet output.

- Start transmission
- Maintain statistics counters

Starting Transmission

As explained above, kernel thread also calls the driver start routine to start transmission. Transmit start routine dequeues a packet from the interface send queue and transmits it.

Statistics Counters

The counters relating to packet output are the `if_opackets`, `if_oerrors` and `if_collisions` fields in the `ifnet` structure. The `if_opackets` counter should be incremented for each packet that is successfully transmitted by the interface without error. If the interface indicates that an error occurred during transmission, the `if_oerrors` counter should be incremented. The driver should also interrogate the interface to determine how many collisions, if any, occurred during transmission of a packet. A collision is not necessarily an error condition. An interface will normally make a number of attempts to send a packet before raising an error condition.

Packet Input

When packets are received by a network interface they must be copied from the device to mbufs and passed to the TCP/IP software. Because this can take a significant amount of time, the bulk of the processing of incoming packets should be done by the driver's kernel thread so that it does not impact the system's real-time performance. The interrupt handler routine should do the minimum necessary to ensure that the interface continues to function correctly.

To maintain bounded system response times, the interrupt handler should also disable further interrupts from the interface. These will be re-enabled by the driver's kernel thread. The processing of input packets involves the following activities:

- Determine packet type
- Copy data from interface into mbufs
- Strip off Ethernet header
- Enqueue packet on input queue
- Re-enable receiver interrupts
- Maintain statistics counters

Determining Packet Type

The packet type is specified in the Ethernet header and is used by the driver to determine where to send the received packet. In the following code fragment, the *ptr* variable is assumed to be a pointer to the start of the received Ethernet frame. The use of the `ntohs` function ensures the portability of code across different CPU architectures.

```
ether_type = ntohs (((struct
        ether_header *)ptr)->ether_type);
```

Copying Data to mbufs

Most network devices have local RAM which is visible to the device driver. On packet reception, the driver must allocate sufficient mbufs to hold the received packet, copy the data to the mbufs, then pass the `mbuf` chain to the TCP/IP software. The `ifnet` structure is added to the start of the packet so that the upper layers can easily identify the originating interface. The Ethernet header must be stripped from the received packet. This can be achieved simply by not copying it into the mbuf(s). If the entire packet can not be copied, any allocated mbufs must be freed. The following code outlines how a packet is copied from the hardware to mbufs using `m_devget` routine. `m_devget` is called with address and size of buffer that contains received packet. It creates a new `mbuf` chain and returns the pointer to the chain.

```
m = m_devget(buf, len, 0, ifp, 0);
```

where *ifp* is the device interface pointer. The variable *buf* points to the received data. This is usually an address in the interface's local RAM.

By default `m_devget` uses `bcopy`, which copies data a byte at a time. A real driver can provide different algorithm for more efficiency and pass its address to `m_devget` routine.

Enqueueing Packet

The packet read routine finally calls a TCP/IP module called `ether_input` to enqueue received packet on one of the TCP/IP software's input queues and for further processing.

```
struct ifnet *ifnet;
struct ether_header *et;
struct mbuf *m;

ether_input(ifp, et, m);
```

Statistics Counters

The counters relating to packet input are the if_ipackets and if_ierrors fields in the ifnet structure. The if_ipackets counter should be incremented for each packet that is successfully transferred from the interface and enqueued on the TCP/IP input queue. Receive errors are normally indicated by the interface in a status register. In this case the if_ierrors counter should be incremented.

Ioctl Entry Point

The ioctl entry point is called by the TCP/IP software with the following syntax:

```
drvr_ioctl (ifp, cmd, arg)
struct ifnet *ifp;
int cmd;                           /* ioctl command id */
caddr_t arg;              /* command specific data */
```

The ioctl function must support the two commands SIOCSIFADDR and SIOCSIFFLAGS.

SIOCSIFADDR

This command is used to set the network interface's IP address. Currently, the only address family supported is Internet. Typically this ioctl gets called by the ifconfig utility. The driver should set the IFF_UP bit in the if_flags and call the drvr_init function to initialize the interface. The argument passed to the ioctl routine is cast to a pointer to an ifaddr structure, which is then used to initialize the interface's Internet address in the arpcom structure. The driver should also call arpwhohas to broadcast its Internet address on the network. This will allow other nodes to add an entry for this interface in their ARP tables.

SIOCSIFFLAGS

This command is used to bring the interface up or down, and is called for example by the command “`ifconfig <name> up`”. The TCP/IP software sets or resets the `IFF_UP` bit in the `if_flags` field before calling the driver’s `ioctl` entry point to indicate the action to be taken. An interface that is down can not transmit packets.

When the interface is being brought up, the driver should call the `drvr_init` function to initialize the interface. When the interface is being brought down, the interface should be reset by calling `drvr_reset`. In both cases the statistics counters in the `ifnet` structure should be zeroed.

The driver normally defines a flag in the statics structure which it uses to keep track of the current state of the interface (`s->ds_flags` in the example code below).

```
struct statics *s;
struct ifaddr *ifa;

case SIOCSIFADDR:
    ifa = (struct ifaddr *) arg;
    ifp->if_flags |= IFF_UP;
    drvr_init (s);
    switch (ifa->ifa_addr->sa_family) {
        case AF_INET :
            ((struct arpcom*)ifp)->ac_ipaddr = IA_SIN
                   (ifa)->sin_addr;
            arpwhohas ((struct arpcom*)ifp, &IA_SIN
                    (ifa)->sin_addr);
             break;
        default :
             break;
    }
    break;

case SIOCSIFFLAGS:
    if ((ifp->if_flags & IFF_UP) == 0 && s->ds_flags &
            DSF_RUNNING) {
        drvr_reset (s);        /* interface going down */
         s->ds_flags &= ~DSF_RUNNING;
    } else if ((ifp->if_flags & IFF_UP) &&
            !(s->ds_flags & DSF_RUNNING)) {
```

```
drvr_init(s);            /* interface coming up */
}
ifp->if_ipackets   = 0 ;
ifp->if_opackets   = 0 ;
ifp->if_ierrors    = 0 ;
ifp->if_oerrors    = 0 ;
ifp->if_collisions = 0 ;
break;
```

watchdog Entry Point

The `watchdog` entry point can be used to implement a function that periodically monitors the operation of the interface, checking for conditions such as a hung transmitter. The function can then take corrective action if necessary. If the driver does not have a watchdog function, the corresponding field in the `ifnet` structure should be set to `NULL` before calling `if_attach`.

The watchdog function is used in conjunction with `if_timer` field in the `ifnet` structure. This field specifies a timeout interval, in seconds. At the expiration of this interval, the TCP/IP module calls the `watchdog` entry point in the driver, passing it the *p* field from the `ifnet` structure as an argument. The `p` field is normally used to contain the address of the statics structure.

Note that the timeout interval specified by `if_timer` is a one-shot function. The driver must reset it to a non-zero value to cause the watchdog function to be called again. Setting the `if_timer` value to 0 will disable the watchdog function.

reset Entry Point

This entry point is called by the kernel during a reboot sequence, passing it the p field from the `ifnet` structure, which is normally the address of the `statics` structure. This function may also be called internally from the driver's `ioctl` entry point. The function should reset the hardware, putting it into an inactive state.

Kernel Thread

The kernel thread receives events from two sources, the interrupt handler (indicating completion of packet transmission or reception) and the driver output routine (indicating the availability of packets on the `if_snd` queue). A single event synchronization semaphore is used for both these purposes. The thread should handle interrupts first and then packets on the output queue. The general structure of the thread will look something like this:

```
struct statics *s;

for (;;) {
    swait (&s->threadsem, SEM_SIGIGNORE);
    handle_interrupts (s);/* handle any interrupts */
    output_packets (s);
                  /* start tranmitter if necessary */
}
```

The precise details of the thread code will be very much dependent on the hardware architecture. The function for processing interrupts will contain the packet input code discussed above. It will also maintain the various statistics counters. Also, receiver interrupts, if disabled by the interrupt handler, are re-enabled at this point. The output function will perform the tasks discussed above in the section on packet output.

Priority Tracking

Whenever the set of user tasks using the TCP/IP software changes or the priority of one of these tasks changes, the `setprio` entry point in the driver is invoked to allow the driver to properly implement priority tracking on its kernel thread.

The entry point is passed two parameters, the address of the `ifnet` structure and the priority which the kernel thread should be set to.

```
drvrsetprio (ifp, prio)
struct ifnet *ifp;
int prio;

{
int ktid;                        /* kernel thread id */

    ktid = ((struct statics *) (ifp->p))->kthread_id;
    stsetprio (ktid, prio);
}
```

Driver Configuration File

The driver configuration file `drvr.cfg`, in the `/sys/lynx.os` directory, needs to declare only the `install` (and `uninstall`) entry points. The other entry points are declared to the TCP/IP module dynamically using the `if_attach` function. A typical configuration file looks something like this:

```
C:wd3e: \
    ::::: \
    :::wd3einstall:wd3euninstall
D:wd:wd3e0_info::
N:wd:0:
```

IP Multicasting Support

ether_multi Structure

For each Ethernet interface there is a list of Ethernet multicast address ranges to be received by the hardware. This list defines the multicast filtering to be implemented by the device. Each address range is stored in an `ether_multi` structure:

```
struct ether_multi {
  u_char enm_addrlo[6];  /* low/only addr of range */
  u_char enm_addrhi[6]; /* high/only addr of range */
  struct arpcom *enm_ac; /* back pointer to arpcom */
  u_int enm_refcount;  /* num claims to addr/range */
  struct  ether_multi *enm_next;
                        /* ptr to next ether_multi */
};
```

The entire list of `ether_multi` is attached to the interface's `arpcom` structure.

1. If the interface supports IP multicasting, the install routine should set the `IFF_MULTICAST` flag. For example:

   ```
   ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST;
   ```

   where `ifp` is pointer to interface `ifnet` structure.

2. Two new ioctls need to be added. These are `SIOCADDMULTI` to add multicast address to reception list and `SIOCDELMULTI` to delete multicast address from reception list:

   ```
   case SIOCADDMULTI:
   case SIOCDELMULTI:
       /* Update our multicast list  */
       error = (cmd == SIOCADDMULTI) ?
       ether_addmulti((struct ifreq *)data,
                  &s->es_ac) :
       ether_delmulti((struct ifreq *)data,
                  &s->es_ac);

       if (error == ENETRESET) {
        /*
         * Multicast list has changed; set the
         * hardware  filter accordingly.
         */
           lanreset(s);
           error = 0;
       }
   ```

3. The driver reset routine must program the controller filter registers from filter mask calculated from multicast list associated with this interface. This list is available in `arpcom` structure and there are macros available to access the list. For example:

```
struct ifnet *ifp = &s->s_if;
register struct ether_multi *enm;
register int i, len;
struct ether_multistep step;

/*
 * Set up multicast address filter by passing
 * all multicast addresses through a crc
 * generator, and then using the high order 6
 * bits as a index into the 64 bit logical
 * address filter. The high order two bits
 * select the word, while the rest of the bits
 * select the bit within the word.
 */

bzero(s->mcast_filter,
            sizeof(s->mcast_filter));
    ifp->if_flags &= ~IFF_ALLMULTI;
    ETHER_FIRST_MULTI(step, &s->es_ac, enm);

    while (enm != NULL) {
        if (bcmp((caddr_t)&enm->enm_addrlo,
            (caddr_t)&enm->enm_addrhi,
             sizeof(enm->enm_addrlo)) != 0) {
 /*
  * We must listen to a range of multicast
  * addresses. For now, just accept all
  * multicasts, rather than trying to set only
  * those filter bits needed to match the
  * range.
  * (At this time, the only use of address
  * ranges is for IP multicast routing, for
  * which the range is big enough to require
  * all bits set.)
  */
            for (i=0; i<8; i++)
                     s->mcast_filter[i] = 0xff;
            ifp->if_flags |= IFF_ALLMULTI;
            break;
```

```
        }
        getcrc((unsigned char *)&enm->enm_addrlo,
              s->mcast_filter);
        ETHER_NEXT_MULTI(step, enm);
    }
```

4. If the driver input routine receives a Ethernet multicast packet, it should set the `M_MCAST` flag in the `mbuf` before passing that `mbuf` to `ether_input`:

```
char *buf;
struct ether_header *et;
u_short ether_type;
struct mbuf *m = (struct mbuf *)NULL;
int flags = 0;

/* set buf to point to start of received frame */
...
...
et = (struct ether_header *) buf;
ether_type = ntohs((u_short) et->ether_type);

if (et->ether_dhost[0] & 1)
    flags |= M_MCAST;

/* pull packet off interface */
...
...
m->m_flags |= flags;
ether_input(ifp, et, m);
```

Berkeley Packet Filter (BPF) Support

1. If system has `bpf` driver installed, call `bpfattach` to register the interface with `bpf`. `bpfattach` is called indirectly via `bpf_attach_p` which is initialized if `bpf` driver is installed

```
if (nbpfilter)
   (*bpf_attach_p)(&ifp->if_bpf, ifp,
   DLT_EN10MB, sizeof(struct ether_header));
```

2. If `bpf` is listening on this interface, let it see the packet before it is committed to the write. This is done by tapping data in the driver output routine as follows:

```
if (ifp->if_bpf)
   (*bpf_tap_p)(ifp->if_bpf, buf, length);
```

   where `buf` is output buffer and `length` is size of packet to be transmitted.

3. If there is a `bpf` filter listening on this interface, hand off the raw (received) packet to `enet`. This is done in the packet receive routine as follows:

```
if (ifp->if_bpf) {
   (*bpf_tap_p)(ifp->if_bpf, sbuf, slen);

/* keep the packet if it is a broadcast or has
 * our physical Ethernet address (or if we
 * support multicast and it is one.
 */
    if ((flags & (M_BCAST | M_MCAST)) == 0 &&
       bcmp(et->ether_dhost, s->s_enaddr,
            sizeof(et->ether_dhost)) != 0)
       return;
    }
```

   where `sbuf` is receiver buffer and `slen` is size of received packet.

4. The `SIOCSIFFLAGS` ioctl should support setting/clearing of promiscuous mode.

```
/*
* If the state of the promiscuous bit changes,
* the interface must be reset.
*/
if (((ifp->if_flags ^ s->es_iflags) &
         IFF_PROMISC) && (ifp->if_flags &
         IFF_RUNNING)) {
    s->es_iflags = ifp->if_flags;
    laninit(s);
}
```

   where **s** is the driver statics structure.

CHAPTER 10

STREAMS Device Drivers

This chapter discusses the general guidelines to writing a STREAMS-based device driver in LynxOS and the steps required to install the device driver on a Lynx system. The call interfaces to all STREAMS module procedures are identical to the STREAMS driver interfaces. Thus, the guidelines mentioned here also apply to the development of a new STREAMS module on LynxOS.

STREAMS Architecture

The kernel code for STREAMS implements the following system calls: `open`, `close`, `ioctl`, `read`, `write`, `putmsg`, `getmsg`, and `poll`. The kernel code also contains a kernel thread for the STREAMS scheduler which executes the service procedures of the enabled queues, and contains the STREAMS kernel utility functions.

In LynxOS, the device drivers are typically implemented as kernel threads which are signalled by interrupt handlers to do the detailed interrupt handling. These device driver threads form STREAMS messages and use the STREAMS utility functions to process the messages. The utility functions are called by user processes executing the STREAMS system calls; the service procedures of drivers/modules invoked by the

STREAMS scheduler; and the STREAMS drivers threads. Any of these processes or kernel threads may be preempted by a higher priority process or thread.

In order to guarantee a good task response time and to maintain consistency of STREAM data structures, a semaphore `str_sem` is used before calling any of the utility functions. It is important that this semaphore is used correctly by the STREAMS device driver threads.

STREAMS Device Driver Entry Points

The STREAMS device drivers differ from non-STREAMS-based drivers in that driver procedures are called from the stream head and not from the system-call entry point. The entry points that a STREAMS driver needs to provide are:

- driver open-time procedure
- driver write-side-put procedure
- driver close-time procedure

The STREAMS driver can also have a read-side-put procedure and a service procedure. But the read-side-put procedure and the service procedure are not directly called by the stream head. The driver puts the pointers to the `open`, `close`, `put`, and `service` procedures in a `streamtab` structure. The pointer to the `streamtab` structure is stored in the `udrivers` table. When the stream head has to call one of the driver's procedures, it takes the address of the procedure from the `streamtab` structure stored in the `udrivers` table. The four general processes that need to execute in the driver's entry points are described in the following sections.

Driver Open-time Processing

Since more than one minor devices can be used for opening a stream, the driver needs to maintain a list of private data structures that it can allocate. At open time, it gets one free data structure for the stream on which the `open` comes. It also needs to maintain linkages between the private data structure and the queue on which the `open` comes, so that all further messages that come downstream can be allocated a data structure to use. Therefore, it has to store the pointer to the private data structure in the queue's `q_ptr`; and the pointer to the queue in its private data structure.

Driver Processing Procedures

The driver processing procedures are broadly categorized into write-side processing procedures and read-side processing procedures.

The write-put-procedure has to handle all the messages that come from top. The procedure can decide whether it can process the messages immediately or not. If for some reason the driver encounters a condition where it cannot process the messages immediately, the driver should have a write-side service procedure also. For example, if the external device cannot accept any more data and blocks the driver from sending any more data, the driver put procedure queues the message in its own write queue. When the service procedure is scheduled, it can check whether the message can then be processed or not, and take appropriate action.

When any data is received from the external device, the driver has to process the data (if necessary) and send it up to the stream head. Since the processing time for doing this can be too costly in the interrupt handler, the interrupt handler queues the message in the driver's read queue and exits. In this case the driver needs to have a read-side service procedure which will be scheduled when the interrupt handler queues the message in the driver's own read queue.

If no processing is required for the data that is received by the driver, the interrupt handler calls the `putnext` routine directly. In this case the driver need not have a read-side service procedure.

Driver Close-time Processing

When a stream is closed, the stream head calls the driver's close procedure before freeing all the STREAMS resources used for this stream. The driver should free private data structure used for this stream. If the driver allows multiple `open` on same stream, it should maintain a reference count in its private data structure and free the data structure only on last `close`.

Handling Flow Control

The STREAMS mechanism, as such, does not provide flow control on any open stream. The STREAMS mechanism provides a set of routines which the driver and module developer should use to handle flow control.

In situations where the driver cannot send any more data to the external device, the user process should be stopped from pumping more data into the stream. When the driver cannot process a message, it should put the message in its own write queue.

Each queue maintains a low-water and high-water mark to be used for flow control. When the number of data bytes queued in the driver's write queue exceeds high-water mark, the stream head stops calling driver's `put` procedure and blocks the user from sending more data downstream.

The flow control mechanism should also take care of the upstream flow of data. When the user process stops receiving data from the stream, all the messages sent up by the driver get queued at the stream head's read queue.

Before putting any message into the stream, the driver should call the `canput` routine to check if more data can be sent up or not. If `canput` for the queue just above the driver's read queue fails, the driver should inform the external device not to send more data. When the backlog is cleared, the driver's read-side service procedure is called. The read-side service procedure can inform the external device that the flow control has been cleared, and to start sending messages upstream.

Handling Allocation Failures

The driver can come across a situation where there are no more buffers left in the STREAM's free buffer pool. Normally, when a data packet is to be received, the interrupt handler should queue an indication in the driver's read queue. When the service procedure is scheduled, it should try to allocate a STREAMS buffer (of required size) to send the data upstream.

If the allocation fails, the driver must first inform the external device that it cannot receive more data. Also, since the indication queued by the interrupt cannot be processed by the driver until a message buffer becomes available, the driver should put the message back in the queue from where it was taken. Then, the driver can use the `bufcall` utility to inquire about the availability of the required-sized buffer.

The `bufcall` utility takes the size of buffer and pointer to a function which has to be called when the buffer of given size becomes available. When the function gets called, it can enable the driver's read queue and normal processing can continue.

Driver Flush Handling

All the STREAMS drivers are expected to incorporate handling of `M_FLUSH` messages. If the `FLUSHW` is set, the write queue message should be flushed. If the `FLUSHR` is set, the read queue should be flushed, the `FLUSHW` bit should be cleared, and the message should be sent upstream using the `qreply` utility function.

Configuring STREAMS into LynxOS Kernels

The driver/module code must be compiled and added into the streams library `/sys/lib/streams.a` as is done for other drivers. If there are any tunable parameters for the driver/module, a `<devinfo.h>` header file (defining a structure with these parameters as its fields) should be added to the `/sys/dheaders` directory. A `devinfo.c` file containing the same structure (with the parameters initialized to desired values) should be added to the `/sys/devices` directory and its `makefile`. This `devinfo.c` file should be compiled and the resulting `devinfo.o` will be archived into the `device .a` archive.

The `/sys/lynx.os` directory, where the kernel is built, contains the file `CONFIG.TBL`. This file contains command lines to control the configuration of device drivers in the kernel. Refer to the *LynxOS Reference Manuals* under “File Formats” for the `config` manual page that describes these command lines.

The STREAMS devices are configured as special character devices. The following example is for configuring a STREAMS device driver sample with two minor devices. The name of the `streamtab` structure for this driver is `sampleinfo`. `sample0` and `sample1` are the node names desired for the two minor devices.

```
C:sample::::::::sampleinfo:
D:sample:::
N:sample0:0::
N:sample1:1::
```

The above lines may also be placed in a file `sample.cfg` and the following line may be added to the `CONFIG.TBL` file:

```
I:sample.cfg
```

The clonable device drivers are specified using the “L” command instead of the “N” command. If `sample` is a clonable device, then the commands are:

```
C:sample::::::::sampleinfo:
D:sample:::
L:sample::
```

The node name for this device is `sample`. Its major device number is that of the clone device. The minor device number depends on the position of this entry in the `CONFIG.TBL`.

STREAMS modules are configured using the “M” command. To configure a sample module with `sample` as name and `sampleinfo` as the `streamtab` structure, the following command line may be placed anywhere in the `CONFIG.TBL` file:

```
M:sample:sampleinfo:
```

The `makefile` in the `/sys/lynx.os` directory invokes the `config` utility to process the `CONFIG.TBL` file. `makefile` generates the `CONFIG.h` file and `nodetab` files. The `CONFIG.h` file contains C code required to configure the specified drivers, devices, and modules. The `nodetab` file contains one line of information for each device node to be created. This file can be processed by the `mknod` utility to create the specified device nodes.

CHAPTER 11

Installation & Debugging

This chapter discusses the two methods of device driver installation in LynxOS: *dynamic* and *static* installation. A brief comparison of the two methods is given to ensure that the systems programmer understands the intricacies involved in each type of installation. This section will help the device driver writer choose the type of installation procedure to suit specific requirements.

Dynamic vs. Static Installation

Dynamic Installation

This method allows a user to install a driver after the operating system is booted. The driver object code is attached to the end of the kernel image and the operating system dynamically adds this driver to its internal structure. A driver installed in this fashion can also be removed dynamically.

The advantages of dynamic installation are:

- Dynamic installation is useful when the device driver is being written. The ease of installation and uninstallation makes it ideal for faster development and debugging. If the kernel has to be rebuilt every time a modification is made to the driver, the development time can become prohibitive. By using dynamic installation this is avoided.

- More than one driver can be used for the same device. If there is a need to use two drivers for the same device, they can be installed according to system needs.
- Memory is not wasted on seldom-used drivers. It is allocated only when needed.

Static Installation

With this method, the driver object code is incorporated into the image of the kernel. The driver object code is linked with the kernel routines and is installed during system start-up. A driver installed in this fashion can be removed (i.e., made ineffective), but its text and data segments remain within the body of the kernel.

The advantages of static installation are:

- With static installation, devices are instantly available on system start-up, simplifying system administration. The initial console and root file system devices must use static installation.
- The installation procedure can be avoided each time the system reboots.

NOTE: *Lynx recommends that during the development of a new driver, dynamic installation be used. The driver can be installed statically once it is working without problems.*

Note that neither method of installation affects a device driver's functionality.

Static Installation

The code organization for static installation is done in the following manner.

Table 11-1: : Code Organization for Static Installation

Directory	File	Description
`/`	`lynx.os`	LynxOS kernel
`/sys/lib`	`drivers.a`	Drivers object code library
	`devices.a`	Device Information declarations
`/sys/dheaders`	`devinfo.h`	Device information definition
`/sys/devices`	`devinfo.c`	Device configuration file
	`Makefile`	Instructions for making `devlib.a`
`/sys/drivers/drvr`	driver source	The source code for the driver `drvr` to be installed
`/sys/lynx.os`	`CONFIG.TBL`	Master device & driver configuration file.
	`Makefile`	Instructions for making `/lynx.os`
`/etc`	`nodetab`	Device nodes
`/sys/cfg`	`driv.cfg`	Configuration file for `driv` driver and its devices.

The following steps describe how to do a static installation:

1. Create a device information definition and declaration. Place your device information definition file `devinfo.h` in the directory `/sys/dheaders` along with the existing header files for other drivers in the system.

2. Make sure that the device information declaration file `<devinfo.c>` is in the `/sys/devices` directory and has the following lines in the file in addition to the declaration.

   ```
   ....
   #include "../dheaders/devinfo.h"
   ....
   ```

   This ensures the presence of the device information definition.

3. Compile the `devinfo.c` file and update the `/sys/lib/devices.a` library file to include `devinfo.o`. This may also be automated by adding `devinfo.xxx` to the `Makefile`.

   ```
       FILES=atcinfo.xxx dtinfo.xxx
   flopinfo.xxx devinfo.xxx
   ```

   To update `/sys/lib/devices.a`, enter:

   `make install`

Driver Source Code

Assuming the new driver is `driver`, the following steps must be followed for driver code installation.

1. Make a new directory driver under `/sys/drivers` and place the code of the device driver there.

2. Create a `Makefile` to compile the device driver.

3. Update the library file `/sys/lib/drivers.a` with the driver object file using the command:

   `make install`

Device and Driver Configuration File

The device and driver configuration file should be created with the appropriate entry points, major device declarations, and minor device declarations. The system configuration file is `CONFIG.TBL` in the `/sys/lynx.os` directory.

The `CONFIG.TBL` file is used with the `config` utility to produce driver and device configuration tables for LynxOS. Drivers, major devices, and minor devices are listed in this configuration file. Each time the system is rebuilt, `config` reads `CONFIG.TBL` and produces a new set of tables and a corresponding `nodetab` file for use with the `mknod` utility.

Configuration File: CONFIG.TBL

The parsing of the configuration files in LynxOS follows these rules.

- The commands are designated by single letters as the first character on a line.
- The field delimiter is `:` .
- Spaces between the delimiter are not ignored. They are treated literally.
- Blank lines are ignored.

The special characters in the configuration file are

`#`	Used to indicate a comment in the configuration file. The rest of the line is ignored when this is found as the first character on any line.
`\`	Used as the continuation character to continue a line even within a comment.
`:`	If the `:` is the first character in the line, it is ignored.
`I:filename`	Used to indicate that the contents of the file `filename` should replace the declaration.

The format of a device driver entry with its major and minor device declarations should look like this:

```
# Character device
C:driver name:driveropen:driverclose: \
    :driverread:driverwrite: \
    :driverselect:driverioctl: \
    :driverinstall:driveruninstall
D:some driver:devinfo::
N:minor_device1:minor_number
N:minor_device2:minor_number

# Block device
B:driver name:driveropen:driverclose: \
    :driverstrategy:: \
    :driverselect:driverioctl: \
    :driverinstall:driveruninstall
D:some driver:devinfo::
N:minor_device1:minor_number
N:minor_device2:minor_number
```

The entry points should appear in the same order as they are shown here. If a particular entry point is not implemented, the field is left out, but the delimiter should still be in place.

If above declarations are in a file `driver.cfg` the entry

```
I:driver.cfg
```

should be inserted into the `CONFIG.TBL` file.

Rebuilding the Kernel

To rebuild the LynxOS kernel, type the following commands:

```
cd /sys/lynx.os
make install
```

In order for the applications programs to use a device, a node must be created in the file system with `mknod`. This can be done automatically by using the `nodetab` file created by `config`.

When the system is rebooted to use the newly-created operating system, the `reboot` command should be given the `N` flag:

```
reboot -aN
```

The **N** flag instructs `init` to run `mknod` and create all the nodes mentioned in the new `nodetab`.

Dynamic Installation

Dynamic installation requires a single driver object file and a pointer to the entry points must be declared. The location of the driver source code is irrelevant in dynamic installation. The installation of the dynamically-loaded device driver need not be done manually. A shell script can be written or a C program can be used to install the device driver after system start-up.

CAUTION! *On x86 and 68k systems, the GNU C compiler cannot be used for dynamic installation of device drivers due to COFF object format issues. You must use the Lynx legacy C compiler (l-cc) instead.*

Driver Source Code

To install a device driver dynamically the entry points must be declared in a structure defined in `<dldd.h>`. The variable should be named `entry_points` and for a block composite driver `block_entry_points` is also required.

The format of the dldd structure is illustrated below:

```
#include <dldd.h>
static struct dldd entry_points = { open, close, read
                                    write, select, ioctl, install,
                                      uninstall, streamtab}
```

For block composite drivers, the block driver entry points are specified as:

```
static struct dldd block_entry_points =
                    { b_open, b_close, b_strategy,
                    ionull, ionull, b_ioctl, b_install,
                      b_uninstall, streamtab}
```

The include file `<dldd.h>` must be included in the driver source code and the declaration must contain the entry points in the same order as they appear above. If a particular entry point is not present in a driver, the field in the dldd structure should refer to the external function `ionull`, which is a kernel function that simply returns `OK`. The last field in the dldd structure, `streamtab`, is only used for STREAMS drivers and should be set to `NULL` for other drivers.

The null device driver which is to be installed dynamically could be written as shown below:

```
/* --------------  NULLDRVR.C -------------------*/

#include <conf.h>
#include <kernel.h>
#include <file.h>
#include <dldd.h>

extern int ionull ();
}
nullread(s, f, buff, count)
char *s;
struct file *f;
char *buff;
register int count;
{
    return 0;
}
nullwrite(s, f, buff, count)
char *s;
struct file *f;
char *buff;
register int count;
{
    return (count);
}
nullioctl()
```

```
{
    pseterr (EINVAL);
    return (SYSERR);
}
nullselect()
{
    return (SYSERR);
}
char *nullinstall()
{
    return (0);
}
static struct dldd entry_points = {
                ionull,
                 ionull,
                nullread,
                nullwrite,
                nullselect,
                nullioctl,
                nullinstall,
                ionull,
                (char *) 0
};
```

Note that calls to a driver entry point replaced by `ionull` will still succeed. If a driver does not support certain functionality, then it must include an entry point that explicitly returns `SYSERR`, as in the case of the ioctl and select entry points in the above example. This will cause calls to these entry points from an application task to fail with an error.

Driver Installation

In this release of LynxOS, follow these recommendations for compiling and installing dynamic device drivers on a specific LynxOS platform.

x86 and 68k

To create dynamically installed drivers on x86 or 68k LynxOS systems, you must use the legacy Lynx C compiler and linker.

1. Compile the driver with the Lynx legacy C compiler:
   `l-cc -o driver.o driver.c`
2. Convert the driver object to COFF format using **`-k`** option of the legacy Lynx linker **`l-ld`**:
   `l-ld -k -r -o driver.obj driver.o`

SPARC

LynxOS for SPARC supports dynamic driver installation with the GNU C compiler. To compile a dynamically loadable device driver, enter:

```
gcc -o driver.obj driver.c
```

PowerPC

LynxOS for PowerPC supports dynamic driver installation with the GNU C compiler but requires a special import file for the linker. The *import file* contains a list of driver service calls used by the device driver. An example is shown in the steps below.

1. Compile the driver with GNU C.
   `gcc -c -o driver.o driver.c`
2. Create an import file. For example, the file `driver.import` contains:

```
sysbrk
iointset
fclear
iointclr
get1page
free1page
```

3. Now create a dynamically loadable object module by entering the following (all on one line):

   ld -bM:SRE -bimport:driver.import
   -o driver.obj driver.o

   The option **-BM:SRE** tells the linker this is a shared reusable module. The option **-bimport** tells the linker the name of the import file.

This will create a object module which has a loader section for dynamic linking and stub functions for kernel callbacks.

All Platforms

Once a dynamic driver object module has been created, this object module can now be dynamically installed.

For character device drivers, enter:

```
drinstall -c driver.obj
```

or

```
drinstall("./driver.obj", CHARDRIVER);
```

For block device drivers, enter:

```
drinstall -b driver.obj
```

or

```
drinstall("./driver.obj", BLOCKDRIVER);
```

If successful, `drinstall` or `dr_install` returns the unique `driver-id` that is defined internally by the LynxOS kernel. For the block composite driver, the `driver-id` returned will be a logical `OR` of the character `driver-id` in the lower 16 bits and the block `driver-id` in the upper 16 bits.

Device Information Definition and Declaration

The device information definition is created the same way as in the static installation. To create a device information declaration a program has to be written to instantiate the device information definition.

Assuming the device information definition appears as:

```
struct my_device_info {
    int address;
    int interrupt_vector;
};

/* myprogram.c */

struct my_device_info devinfo = { 0xd000, 4 };

main()
{
    write(1, &devinfo, sizeof(struct \
        my_device_info));
}
```

This program can be compiled and executed while redirecting the output to a file. The compilation of `myprogram.c` is

```
gcc -o myprogram myprogram.c
```

The device information file can then be created using:

```
myprogram > mydevice_info
```

Device Installation

The installation of the device should be done after the installation of the driver. The two ways of installing devices are either through the `devinstall` utility program or `cdv_install` and `bdv_install` system calls. (See the *LynxOS Reference Manuals*, under “Utility Programs & System Calls”).

```
devinstall -c -d driver_id mydevice_info
devinstall -b -e raw_driver_id -d
block_driver_id\mydevice_info
```

The `driver_id` is the identification number returned by the `drinstall` command or system call. This installs the appropriate device with the corresponding driver and assigns a major device number to it (in this case, we’ll assume this is `major_no`).

Node Creation

Unlike the static installation, there is no feature to automatically generate the nodes under dynamic installation. This should be done manually using the `mknod` command. (See the *LynxOS User's Guide*).

The creation of the node should be done in the `/dev` directory for the sake of consistency. The creation of the nodes allows application programs to access the driver by opening and closing the file that has been associated with the driver through the `mknod` command.

mknod /dev/*device* **c** *major_no minor_no*

The *major_no* is the number assigned to the device after a `devinstall` command. This can be obtained by using the `devices` command. The *minor_no* is the minor device number which can be specified by the user in the range of 0–255. The **c** indicating a character device could also be a **b** to indicate a block device.

Device and Driver Uninstallation

Dynamically loaded device drivers can be uninstalled when they are no longer needed in the system. This can help in removing unwanted code in physical memory when it is no longer relevant. Removal is performed with the `drinstall` command. However, the device attached to the driver has to be uninstalled before uninstalling the driver. Removing the device is accomplished with the `devinstall` command.

For character devices:

```
devinstall –u –c device_id
```

For block devices:

```
devinstall –u –b device_id
```

After the device is uninstalled the driver can be uninstalled using the command:

```
drinstall –u driver_id
```

Common Error Messages During Dynamic Installation

The following list describes some common error messages that you may encounter during dynamic installation of a device driver. There also will be a number of messages printed to the system console by the LynxOS kernel that may help your debugging efforts.

- Bad Exec Format

  This is usually seen when a `drinstall` command is executed. If this happens it indicates that a symbol in the device driver has not been resolved with the kernel symbols. You should make sure that there are no symbols which cannot be resolved by the kernel. Also make sure that the structure `dldd` has been declared inside the driver.

- Device Busy

  This error message is seen when you try to uninstall the driver before uninstalling the device. The order has to be first to uninstall the device and then uninstall the driver.

NOTE: *You cannot dynamically install a driver on kernel that has been stripped.*

Debugging

The development of device drivers can become complicated unless proper debugging facilities are available. Since device drivers are not attached to a particular control terminal, ordinary `printf()` statements will not work. LynxOS provides a device driver service routine to debug device drivers effectively. The routine is `kkprintf()`. This routine has exactly the same format as the `printf` call but the output is always sent to the debug terminal which is configured as `com2` on the x86 and `tty0` on the MVME167. Currently the debug terminals are not configurable.

The device driver support routine `cprintf` will print to the console. Unlike `kkprintf`, `cprintf` cannot be used in an interrupt routine or where interrupts or pre-emption is disabled.

The `kkprintf` routine can be used in interrupt routines also as it polls the UART for it to be ready for the next character.

The Simple Kernel Debugger (SKDB) can also be used to debug device drivers. It allows breakpoints to be set and can display stack trace and register information. See the *LynxOS User's Guide* for more information on SKDB.

NOTE: *You cannot use SKDB to debug dynamically installed drivers.*

Hints for Device Driver Debugging

- Insert some `kkprintf` statements in the `install` routine of the new driver. After the `devinstall` command is executed, the `install` entry point is invoked. This is a good way of identifying if the `install` entry point is being invoked at all and if the device information declaration passed is being received properly.
- Insert `kkprintf` statements in the `uninstall` entry point. After every de-installation the `uninstall` entry point is invoked. The `kkprintf` statements should be seen after de-installation.
- Initially, it is advisable to put a `kkprintf` statement at the beginning of every entry point to make sure it is being invoked properly. Once all the errors have been found, these can be removed.

Additional Notes

- Statically installed device drivers in LynxOS can also be uninstalled dynamically. However the memory reclamation is not done in this case.
- Symbols from two or more dynamically-loaded device drivers cannot be resolved. If there are two dynamically-loaded device drivers using function `f()`, the code for function `f()` has to be present in both the drivers' source code. This is because if one of the drivers is loaded initially, function `f()` does not get resolved with the second device driver even though it is in memory. Thus, only statically-loaded drivers' symbols are resolved with dynamic drivers.
- Garbage collection is not provided in LynxOS. Thus, any memory which is dynamically allocated must be freed before uninstalling the device driver.

APPENDIX A

Porting UNIX Drivers to LynxOS

This appendix discusses the similarities and differences between device drivers under LynxOS and UNIX. It is intended to serve two main purposes:

- To provide some guidelines to developers wishing to port existing UNIX drivers to LynxOS in order to reuse existing code.
- To provide a pedagogical stepping stone for those developers who are already experienced with UNIX drivers.

The sections below take a feature of a UNIX device driver as a starting point, then point out the corresponding feature in a LynxOS device driver. This appendix supplements the more detailed coverage of LynxOS device drivers that can be found in the previous chapters of this manual. Certain LynxOS features, not used in a UNIX driver, are consequently only mentioned very briefly. The versions of the UNIX kernel referred to in this chapter are for the most part SVR3.2 and SVR4.

Kernel and Driver Design Goals

A frequently asked question is whether it would be possible to achieve source level or even binary compatibility between UNIX and LynxOS drivers. While this might with some effort be technically feasible, the result would probably not be acceptable for designers of real-time systems.

The reason for this is due to the fundamental differences in the design goals of LynxOS compared to the UNIX kernel. The latter was designed for multi-user, time-sharing systems while LynxOS was designed specifically for hard real-time systems. These differences in design goals influence the choice of kernel data structures and algorithms, including those used in device drivers.

The differences are also seen in the services provided by the kernel to device drivers. The LynxOS kernel provides many services that meet the specific requirements of real-time systems. These features would not be found in a UNIX driver. On the contrary, a UNIX driver may use some services that would result in a detrimental effect on real-time performance.

Another significant difference is pre-emptibility. The UNIX kernel was originally written to be uninterruptible, though some UNIX kernels now exist that are pre-emptible to a greater or lesser extent. The LynxOS kernel, including device drivers, is fully pre-emptible and this has a major influence on the way a driver is written.

Different design goals can also be noted at the level of drivers themselves. UNIX drivers are generally designed to make the most efficient use of I/O devices, thereby maximizing throughput. This goal leads to the use of specific driver techniques such as the chaining of I/O requests, processing of interrupts within an interrupt handler and the starting of the next I/O operation from within the interrupt handler. In contrast, a LynxOS driver must be designed to have a minimal impact on real-time performance, respecting the relative priorities of the tasks that are using the devices. The way in which interrupts are handled is probably the largest difference between a UNIX and a LynxOS driver.

Given these differences, both at the kernel and driver level, it is clear that in order to respect real-time demands, a port is preferable to providing compatibility.

Porting Strategy

Porting a UNIX driver can be broadly divided into three stages, which deal with the following topics:

Stage 1	driver interface with kernel
	driver interface with application
	the u structure
	re-entrancy and synchronization
Stage 2	system threads and priority tracking
Stage 3	dynamic installation
	POSIX programming model

The first stage allows the developer to reach a point where he has a working LynxOS driver which can be tested from a functional point of view. While enabling the re-use of a driver in a relatively short time, this initial port does not take advantage of the real-time aspects of LynxOS and the driver could have a detrimental effect on system response time. In order for the driver to conform to the real-time characteristics of LynxOS, implementation of stage two is absolutely necessary. The features in stage three are optional but may be advantageous in certain situations.

Driver Structure

Overall Structure

A LynxOS and a UNIX driver are quite similar in their overall structure. Each consists of a number of entry points, including an initialization routine, and an interrupt handler. A LynxOS driver has in addition one or more kernel threads.

UNIX	LynxOS
initialization	initialization
entry points	entry points
interrupt handler	interrupt handler
	kernel thread(s)

Global Variables

A UNIX driver typically makes widespread use of global variables which is often the only way for the driver entry points to share information. A LynxOS driver can and should be written without the use of any global variables. The LynxOS kernel provides an elegant means to communicate driver state between entry points. Use of this mechanism is essential to allow the dynamic installation and deinstallation of a driver.

Major and Minor Device Numbers

There is an important difference in the way UNIX and LynxOS use major device numbers. Under UNIX, the major device number is used to distinguish between different drivers. The minor number distinguishes between different devices controlled by the same driver. Under LynxOS, each driver has a unique driver ID, though this number is never used by the driver code. Different devices controlled by the same driver are identified by different major numbers (as opposed to minor number under UNIX). The use of the minor device number is defined entirely by the driver. LynxOS driver IDs and major numbers are allocated automatically during a kernel build though some changes to the driver code will be necessary.

Driver Interface With Kernel

The interface between the UNIX kernel and a driver is defined by the driver service calls, the `init` entry point and the interrupt handler.

Driver Service Calls

The services provided by a kernel to device drivers can be grouped into several functional classes:

- Memory Management
- Synchronization
- DMA Transfers and Raw I/O
- Block I/O
- Driver Debugging

Memory Management

This section describes the functions used for allocating memory and for translating memory addresses.

Memory Allocation

Functions used for the allocation of memory for driver internal use.

UNIX	LynxOS
`kmem_alloc, kmem_free`	`sysbrk, sysfree`
	`get1page, free1page`
	`alloc_cmem, free_cmem`

The functions `sysbrk`/`sysfree` are the nearest equivalent to UNIX `kmem_alloc`/`kmem_free`. The UNIX function `kmem_alloc` can sleep while waiting for free space. The LynxOS functions never sleep; instead, they return `SYSERR` if the memory request can not be satisfied immediately.

Address Translation

The following list shows the functions for converting between virtual and physical addresses.

	UNIX	LynxOS
User virtual to physical	`vtop`	`mmchain, mmchainjob`
Kernel virtual to physical	`kvtophys`	`mmchainjob (job 0)` `addr - PHYSBASE`

Note that `mmchain` returns a kernel virtual address. To convert this to a physical address, the constant `PHYSBASE` must be subtracted.

	UNIX	LynxOS
Number of bytes to number of pages	`btop`	`bytes >> PAGESHIFT`
(rounded up)	`btopr`	`(bytes + PAGESIZE - 1)>>PAGESHIFT`
Number of pages to number of bytes	`ptob`	`pages << PAGESHIFT`

Synchronization

In non-preemptible UNIX kernels, synchronization is a fairly straightforward matter. But in a fully pre-emptible kernel such as LynxOS it is much more complex. This can represent a significant proportion of the porting effort. For more information, see Chapter 4, “Synchronization”

DMA Transfers and Raw I/O

Setting up DMA transfers requires the following kernel services:

- memory locking
- split transfer into physically contiguous pieces
- virtual to physical address translation

The following code fragments illustrate typical SVR4 driver code for performing DMA a transfer to user space.

UNIX

```
read (dev, uio)
dev_t dev;
struct uio *uio;
{
    physiock (mybreak, 0, dev, B_READ, nblocks, uio);
}

mybreak (bp)
struct buf *bp;
{
    dma_pageio (mystrategy, bp);
}

mystrategy (bp)
struct buf *bp;
{
    physaddr = vtop (bp->b_addr, bp->b_proc);
    /* start DMA transfer */
}
```

The key functions in previous code fragment are:

physiock faults in and locks memory pages.

dma_pageio	breaks transfer into 512 byte blocks. Calls strategy routine.
mystrategy	(user written) converts user virtual address to physical address, sets up and initiates DMA transfer.

LynxOS

```
read (s, f, buff, count)
struct statics *s;
struct file *f;
char *buff;
int count;
{
    struct dmachain *array;
    int np, nc, pid, i;

    pid = getpid ();
    np = npages (buff, count);
    array = (struct dmachain *) sysbrk (np * sizeof
              (struct dmachain));
    mem_lock (pid, buff, count);
    nc = mmchain (array, buff, count);
    for (i = 0; i < nc; i++) {
        /*
         * Do DMA transfer at physical address
         * array[i].address, length array[i].count
         */
        array[i].address -= PHYSBASE;
                        /* convert to physical address */
      do_dma (&array[i]);   /* user supplied routine */
    }
    sysfree (array, np * sizeof (struct dmachain));
    mem_unlock (pid, buff, count, TRUE);
}
```

The key functions in previous code fragment are:

mem_lock	faults in and locks pages.
mmchain	converts virtual address range to list of kernel virtual addresses. These are converted to physical addresses using PHYSBASE.

mem_unlock unlocks memory pages.

Note that whereas UNIX uses the block interface (strategy entry point) for raw I/O, LynxOS uses the character interface read and write entry points.

Block Input/Output

Strategy Entry Point

Both UNIX and LynxOS block drivers have a strategy entry point that is called by the kernel's block buffering I/O subsystem to perform transfers to block devices.

UNIX	**LynxOS**
`strategy (bp)`	`strategy (s, bp)`
`struct buf *bp;`	`struct statics *s;`
	`struct buf_entry *bp;`

As with other entry points, the LynxOS strategy routine is passed the address of the devices statics structure as the first argument.

The buf Structure

This data structure defines the buffers that are used to hold the data blocks from a block device. In LynxOS, this structure is of type `struct buf_entry`. The correspondence between the fields is shown below.

UNIX `struct buf`	**LynxOS** `struct buf_entry`
`int b_flags`	`int b_status`
`struct buf *av_forw`	`struct buf_entry *av_forw`
`struct buf *av_back`	`struct buf_entry *av_back`
`o_dev_t b_dev`	`int b_device`
`unsigned b_count`	
`caddr_t b_addr`	`char *memblk`
`daddr_t b_blkno`	`long b_number`

UNIX `struct buf`	**LynxOS** `struct buf_entry`
`char b_oerror`	
`unsigned int b_resid`	
`clock_t b_start`	
`struct proc *b_proc`	
`struct page *b_pages`	
`long b_bufsize`	
`int (*b_iodone)()`	
`struct vnode *b_vp`	
`int b_error`	
`dev_t b_edev`	

The symbolic constants used to specify bits in the *b_flags* field are shown below.

UNIX	**LynxOS**
`B_BUSY`	`B_BUSY`
`B_DONE`	`B_DONE`
`B_ERROR`	`B_ERROR`
`B_PAGEIO`	
`B_PHYS`	`B_PHYS`
`B_READ`	`B_READ`
`B_WANTED`	`B_WANTED`
`B_ASYNC`	`B_ASYNC`

Block I/O Support Routines

UNIX provides a number of support routines for block device drivers.

`biowait`	suspend, waiting for I/O completion
`biodone`	wakeup process and release buffer
`brelse`	put buffer back on free list

The following code fragment shows how these routines are typically used in the strategy entry point and interrupt handler of a UNIX driver.

UNIX

```
xx_strategy (bp)
struct buf *bp;
{
    /* start transfer on device ... */
    ....
    /* if transfer is asynchronous, return, else wait
       for completion */

    if (bp->flags & B_ASYNC)
        return (0);
    biowait (bp);
}

xx_intr ()
{
    if (error_condition)
        bp->b_error |= B_ERROR;
   biodone (bp);                 /* wake up process */
}
```

LynxOS does not provide the `biowait` or `biodone` routines, but the code to implement the required functionality is straightforward, as shown below.

LynxOS

```
strategy (s, bp)
struct statics *s;
struct buf_entry *bp;
{
    /* start transfer on device ... */

   swait (&s->devsem);  /* wait for device completion
*/
    bp->b_status |= B_DONE;
             /* set bits to indicate transfer status */
    if (s->error)
        bp->b_status |= B_ERROR;
    if (bp->b_status & B_ASYNC) {
            /* if async transfer, release buffer ... */
        ssignal (&bp->b_rwsem);
        brelse (bp);
    } else
        ssignal (&bp->b_rwsem);
                    /* ... else wakeup waiting task */
}
```

Driver Debugging

UNIX

`printf`	print message on system console (uses polling)
`uprintf`	print message on user terminal (uses driver)

LynxOS

`kprintf`	print message on debug console (uses polling)
`cprintf`	print message on system console (uses driver)

Initialization Routine

Although both UNIX and LynxOS drivers have an initialization routine, the way in which they are used differs in some important ways. By convention, the UNIX routine is called *xxx*`init`, in LynxOS *xxx*`install`.

UNIX

- Initialization is called once during boot up.
- Initializes all hardware and software.
- Device specific information kept in statically allocated structures.
- Maximum number of supported devices is hard coded.
- Limited number of configuration parameters.

LynxOS

- Initialization routine is called for every major device.
- Device structure allocated dynamically.
- Number of supported devices not limited.
- User definable configuration parameters.

Probing For Devices

One of the tasks usually performed by the initialization routine is to test for the presence of a device. UNIX drivers must handle bus errors. In LynxOS this is handled automatically. Typical UNIX and LynxOS code is illustrated below:

UNIX init routine

```
#define MAX_CONT 4 /* no. of supported controllers */
struct csb csb[MAX_CONT]; /* controller status blocks
*/

xx_init ()
{
    for (i = 0; i < xx_ccnt; i++) {
        if (setjmp (u.u_tsav) == 0) {
            u.u_nofault = TRUE;
          ......           /* touch the device here */
            u.u_nofault = FALSE;
         ......  /* Initialize hardware and software */
        } else
          xx_addr[i] = 0;      /* device not present */
    }
}
```

LynxOS install routine

```
xx_install (info)
struct xxinfo *info;
           /* user defined configuration parameters */
{
    ....   /* Touch device here */
        /* If we get here, we know device is present */
    s = (struct statics *) sysbrk (sizeof (struct
            statics s));
    ....   /* Initialize software and hardware */
    return (s);
}
```

Interrupt Handling

In UNIX System V, the details of a device's interrupt capabilities are defined statically in configuration files external to the driver. The name of the interrupt handler is *xxx*_`intr`, where *xxx*_ is the specified driver prefix.

Since LynxOS supports dynamic driver installation and deinstallation, attaching and detaching an interrupt handler is done within the driver code using the functions `iointset` and `iointclr`. This is done in the `install` and `uninstall` entry points. The device's interrupt vector is normally passed to the install routine in the device information structure.

For x86

```
iointset (32 + info->vector, intr_handler, s);
```

For 68k

```
iointset (info->vector * 4, intr_handler, s);
```

The U Structure

Unlike most UNIX kernels, LynxOS does not have a `u` structure. The following paragraphs discuss the most commonly used members of this structure and how the equivalent functionality is implemented in a LynxOS driver.

u_base, u_count, u_offset

Older versions of UNIX used these fields to specify the details of a data transfer in the read/write entry points. The driver modifies these during the course of the transfer. The return value received by the application is the initial `u_count` value minus its final value. More recent implementations of UNIX have replaced them with a `uio` structure.

In a LynxOS driver, the user buffer address and size are passed directly as arguments to the driver entry point. An important difference from UNIX is that the value returned to the application is the value returned by the driver entry point. The seek position on the device is specified by the field *position* in the file structure. The driver is responsible for setting this at the end of a transfer.

u_fmode

Holds the file mode flags. Its main use is in the read/write entry points to test for non-blocking I/O. It is also used to test, for example, if an application is trying to read from a device opened in write only mode.

In LynxOS, the file mode is held in the *access_mode* field of the file structure.

u_error

This field contains an error code which is copied to the application's `errno` variable.

A LynxOS driver specifies an error code with the `pseterr` function.

u_segflg

Indicates whether a data transfer is to/from user or kernel space. It is necessary to know this because the user process and kernel have separate virtual address spaces.

In LynxOS, the user process and kernel exist within the same virtual space, so this functionality is not required.

u_procp

Pointer to process table entry for current process. UNIX device drivers seldom need to access this field explicitly. In LynxOS, each process is identified by a unique job number which can be accessed in the driver top half routines to provide similar functionality. The function `getpid` can also be used to find the process id number.

u_tsav, u_nofault

Used for trapping bus errors, typically in the `init` routine.

In the install routine of a LynxOS driver, bus errors are handled automatically. Elsewhere in a driver, the routines `noreco`/`recoset` must be used to catch bus errors.

Re-entrancy and Synchronization

Critical Code Regions

Accesses to shared data structures and hardware registers must be serialized. The synchronization mechanisms used in a UNIX driver depend very much on whether the driver is pre-emptible. SVR4 driver code is not pre-emptible though synchronous pre-emption is possible if a driver calls *sleep*. Drivers written for such kernels only need to synchronize with the interrupt level routines. This is done with the spl*n* and splx functions. The LynxOS equivalent of these functions is `disable`/`restore` although there is an important difference. The LynxOS functions `disable`/`restore` all interrupts, interrupt nesting is not possible.

Drivers under LynxOS are fully pre-emptible. Appropriate synchronization must be added to make the driver re-entrant.

Event Synchronization

This type of synchronization involves waiting for an event (buffer free, transfer complete, data ready etc.) to occur.

UNIX	LynxOS
`sleep/wakeup`	`swait/ssignal`

The UNIX `sleep` function specifies a priority which is assigned to the process when it wakes up. LynxOS uses fixed scheduling priorities. A task priority can only be changed on request from the user application. Both UNIX sleep and LynxOS `swait` use an argument to specify how signals are handled during the time the task is blocked. It is difficult to find an exact correspondence in behavior in all cases.

<table>
<tr><th>UNIX sleep priority</th><th></th></tr>
<tr><td><= PZERO</td><td>Signals are ignored. Use the symbolic constant SEM_SIGIGNORE with swait.</td></tr>
<tr><td>> PZERO</td><td>Signals are delivered but sleep never returns. The nearest equivalent with swait is to use the symbolic constant SEM_SIGRETRY. However, the swait is automatically restarted and will eventually return.</td></tr>
<tr><td>> PZERO | PCATCH</td><td>sleep is aborted and returns 1 on receipt of a signal. The LynxOS equivalent is to use the symbolic constant SEM_SIGABORT. However, swait returns a non-zero value (not necessarily 1).</td></tr>
</table>

Another important difference is that wakeup is stateless. It can only wakeup tasks that are blocked on the event at the time that wakeup is called. On the other hand, ssignal has a counter associated with it. This difference can have an influence on driver design. More care is needed with synchronization in the stateless case (though this problem is normally solved by the fact that a UNIX driver is not pre-emptible!).

Driver Interface with User Applications

The driver interface with the application covers the following topics:

- Driver Entry Points
- Accessing User Address Space
- Returning Errors

Entry Points

There are a number of general remarks that can be made that apply to all entry points.

- In a LynxOS driver the first argument to all entry points is a pointer to the `statics` structure allocated by the install routine.
- LynxOS does not use the UNIX `cred_t` credentials structure.
- In LynxOS, the device number is passed only to the open entry point. Other entry points can access the device number and access mode in the `file` structure.

```
int flag = f->access_mode;
int dev = f->dev;
```

Major and Minor Device Numbers

As discussed above, there is an important difference in the way UNIX and LynxOS use the device numbers. Typically, a UNIX driver uses (part of) the minor number to index into an array containing state variables for each device, as illustrated below.

UNIX

```
/* number of supported controllers */
#define MAX_CONT 4

/* controller status blocks */
struct csb csb[MAX_CONT];

/* number of configured controllers */
extern int xxx__ccnt;

xxx_open (dev, mode, otyp, cred)
dev_t *dev;
int mode;
int otyp;
cred_t *cred;
{
    struct csb *csbp;
    int cntlr;

    cntlr = getminor (*dev) & 0xf;
    if (cntlr >= xxx__ccnt || cntlr >= MAX_CONT)
        return (ENXIO);
    csbp = &csb[cntlr];
```

This code is unnecessary in the LynxOS driver because the address of the controller's status block is passed as an argument to the entry points.

LynxOS

```
xxx_open (s, dev, f)
struct statics *s;
int dev;
struct file *f;
```

NOTE: UNIX *drivers usually use the term controller status block. LynxOS drivers use a statics structure. They are more or less the same thing!*

open/close

UNIX	LynxOS
`open (dev,mode,otyp,cred)`	`open (s, dev, f)`
`dev_t *dev;  /* SVR4 */`	`struct statics *s;`
`int mode;`	`int dev;`
`int otyp;`	`struct file *f;`
`cred_t *cred;`	
`close (dev,mode,otyp,cred)`	`close (s, f)`
`dev_t dev;`	`struct statics *s;`
`int mode;`	`struct file *f;`
`int otyp;`	
`cred_t *cred;`	

As shown in the listing above:

- LynxOS passes the device number to open, like SVR3. SVR4 passes a *pointer* to the device number.
- LynxOS does not have an equivalent of the `otyp` field.
- The LynxOS kernel only calls the close entry point on the *last* close of a device.

read/write

UNIX	LynxOS
`read (dev, uiop, credp)`	`read (s, f, buff, count)`
`dev_t dev;`	`struct statics *s;`
`uio_t *uiop;`	`struct file *f;`

UNIX	LynxOS
`cred_t *credp;`	`char *buff;` `int count;`

The UNIX `uio` structure specifies a list of user buffers. Earlier UNIX kernels used the `clist` data structure for character storage.

In a LynxOS driver, the user buffer is specified by *buff* and *count*. The entry point is called once for each buffer in scatter/gather I/O (`readv`, `writev`). LynxOS does not use the `clist` data structure.

The following code fragments compare typical write entry point logic used to transmit all the user data. Note that in LynxOS, the driver is responsible for positioning the seek pointer (`f->position`). Another important difference is that the UNIX driver returns the number of bytes *not* transmitted.

UNIX

```
while ((c = uwritec (uio))
>=0)
    transmit (c);
return (0);
```

LynxOS

```
for (i = 0; i < count;
i++)
    transmit (buff[i]);
    f->position += count;
return (count);
```

ioctl

UNIX	LynxOS
`ioctl (dev, cmd, arg, mode, cred, rval)`	`ioctl (s, f, cmd, arg)`
`dev_t dev;`	`struct statics *s;`
`int cmd;`	`struct file *f;`
`int arg;`	`int cmd;`
`int mode;`	`int arg;`
`cred_t *credp;`	
`int *rval;`	

If *arg* is a pointer, the LynxOS driver must check the validity of the address with *rbounds* and *wbounds*.

select

UNIX	LynxOS
`extern int selwait;`	`struct statics {`
`struct proc *selr, *selw;`	`...`
	`int space_free; /* for output */`
	`int data_ready; /* on input */`
	`int *rsel_sem, *wsel_sem;`
	`};`
`select (dev, rw)`	`select (s, f, which, ffs)`
`dev_t dev;`	`struct statics *s;`
`int rw;`	`struct file *f;`
`{`	`int which;`
`switch (rw) {`	`struct sel *ffs;`

UNIX	LynxOS
`    case FREAD:`	`{`
`        selr = u.u_procp;`	`switch (which) {`
`        break;`	`    case SREAD:`
`    case FWRITE:`	`        ffs->iosem = &s->data_ready;`
`        selw = u.u_procp;`	`        ffs->sel_sem = &s->rsel_sem;`
`        break;`	`        break;`
`  }`	`     case SWRITE:`
`  return(0);`	`        ffs->iosem = &s->space_free;`
`}`	`        ffs->sel_sem = &s->wsel_sem;`
	`        break;`
	`    case SEXCEPT:`
	`        return (SYSERR);`
	`}`
	`return(OK);`
	`}`
`/*Data Input */`	
`if (selr) {`	`s->data_ready = 1;`
`        selwakeup (selr, coll);`	`disable (ps);`
`        selr = 0;`	`if (s->rsel_sem)`
`}`	`        ssignal (s->rsel_sem);`
	`restore (ps);`
`/* Data Output */`	
`if (selw) {`	`s->space_free = 1;`
`        selwakeup (selw, coll);`	`disable (ps);`
`        selw = 0`	`if (s->wsel_sem)`
`}`	`        ssignal (s->wsel_sem);`
	`restore (ps);`

Accessing User Space

UNIX

The currently executing user process and the kernel may have separate virtual address spaces. In this case, kernel service routines are used to transfer data to and from user space. These routines usually handle invalid user addresses.

LynxOS

The current user process and the kernel exist in same virtual space. The kernel (including drivers) can access the whole of the virtual space. Therefore, drivers can transfer data to and from user space directly using a pointer.

The following code fragments illustrate how data might be transferred from user space in an `ioctl` entry point.

UNIX	LynxOS
`char *useraddr, *kernaddr;` `int nbytes;` `if (copyin (useraddr, kernaddr, nbytes) == -1)` `    return (EFAULT);`	`if (rbounds (useraddr) < nbytes) {` `    pseterr (EFAULT);` `    return (SYSERR);` `}` `while (nbytes--)` `*kernaddr++=*useraddr++;`

Returning Errors to User Application

UNIX

Earlier versions used the `u_error` field in the `u` structure. SVR4 uses the entry point return value.

LynxOS

Use the `pseterr` function and return the value `SYSERR`.

The following code fragments illustrate how a driver would return the error `EIO`:

UNIX SVR3	UNIX SVR4	LynxOS
`u.u_error = EIO;`	`return (EIO);`	`pseterr (EIO);`
`return;`		`return (SYSERR);`

LynxOS Kernel Threads

Using kernel threads, interrupt processing is performed by a pre-emptible, prioritized task. This is essential in order to maintain deterministic system response times. Using the UNIX interrupt architecture where all interrupt processing is done in the interrupt handler itself will lead to a degradation of the system's real-time performance.

Dynamic Installation

LynxOS supports the dynamic installation and deinstallation of drivers. This greatly facilitates the driver development and debugging phases as a kernel rebuild and reboot is not necessary each time the driver is modified. If the port has been done correctly, the only additional required to support dynamic installation is the declaration of the `entry_points` structure.

POSIX Programming Model

The LynxOS implementation of the POSIX.1 and POSIX.1b features permits a much simpler driver design for supporting asynchronous I/O, non-blocking I/O and synchronous I/O multiplexing and polling.

Asynchronous I/O

The complexity of handling asynchronous transfers is hidden from the application and driver developer. The POSIX API provides services to the application developer and the driver sees only synchronous requests. Therefore, code to handle asynchronous transfers can be removed from a UNIX driver if the LynxOS version is only intended for use with POSIX conforming applications.

Synchronous I/O Multiplexing and Polling

This functionality is provided by the select system call at the application level and the select entry point in a driver. The POSIX standard does not define a select function. So, if the LynxOS driver is only intended for use with POSIX conforming applications, the select entry point can be removed.

Index

A

B

C

D

E

P

Q

R

S

T

U

V

W