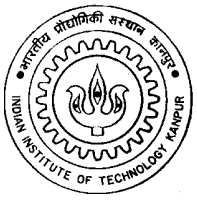
Userdev: A Framework For User Level Device Drivers In Linux

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Hari Krishna Vemuri



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April 2002

Certificate

This is to certify that the work contained in the thesis entitled " *Userdev: A Framework For User Level Device Drivers In Linux* ", by *Hari Krishna Vemuri*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

April 2002

(Dr. Deepak Gupta and Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Abstract

Device drivers have always been complex software that are written at the operating system level. They are knit with the rest of the operating system kernel and work by interacting with it. Writing device drivers is a cumbersome task because writing and debugging the kernel code is very difficult. It is thus desirable to have a mechanism for implementing user level device drivers.

In this thesis we describe the design and implementation of the *Userdev framework* that allows implementation of user level device drivers in Linux. The user level device drivers written using this framework present exactly the same interface to the applications as a kernel driver and thus existing applications that use the device can be run without any change, re-compilation or even re-linking. We have also developed prototype user level device drivers for a parallel port line printer, a floppy drive and a RAM disk to demonstrate the feasibility of our model. Experiments show that these user level device drivers perform almost as efficiently as their kernel counterparts.

Acknowledgement

I take this opportunity to express my sincere gratitude to my supervisors Dr. Rajat Moona and Dr. Deepak Gupta for their invaluable guidance. It would not have been possible for me to take this project to completion without their relentless support and encouragement. I consider myself extremely fortunate to have had a chance to work under their supervision. It has been a very enlightening and enjoyable experience to work under them.

I also wish to thank all the faculty members of the Department of Computer Science and Engineering for imparting their invaluable knowledge in course of my Mtech program. I also extend my thanks to the technical staff of the department for maintaining an excellent working facility.

I would also like to thank my batchmates who have made my stay in IIT Kanpur, the most memorable one. It was as though i was a part of a big family, studying, working and enjoying together. The 'mtech2000' mailing list was the most favourite email address which hosted online discussions on a wide range of topics. The weekend outings which provided the required break from studies, formed an unforgettable part of my hostel life.

Finally I would like to thank my parents and sister for providing the necessary support and encouragement for building a good career and a bright future.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Device drivers are usually a part of the operating system kernel and have complex interactions with the kernel. Therefore writing and debugging a device driver is a very cumbersome task. In this thesis we present the design and implementation of the *Userdev framework* that allows implementation of device drivers as user level processes. User level device drivers are clearly easier to write than the kernel drivers since understanding of the kernel internals is not necessary for developing a user level driver. Debugging is also easier since standard debugging tools can be used and a crash of the driver does not bring down the system.

The disadvantage of user level device drivers is clearly a loss in performance due to overheads of context switching etc. However for a fast and performance critical device, a user level driver can be used as a quickly built prototype that can be later ported to the kernel. For a slow device, the slight loss in performance is likely to be acceptable.

User level device drivers can also be useful for providing drivers for "pseudo-devices" such as pseudo terminals in Linux. Another interesting use of user level drivers is to implement transparent access to remote devices. In this case the device driver would implement the client side of some remote device access protocol. It is clearly not desirable to implement such protocols in the kernel.

The rest of the thesis is organized as follows. Chapter 2 mentions some of the work related to this thesis. Chapter 3 provides some background about Linux device interface and the structure of the kernel device driver. Chapter 4 describes the architecture of the Userdev framework and provides details about the Userdev driver and the Userdev library. Next, chapter 5 gives a brief description of the prototype drivers that have been implemented. Chapter 6 discusses the performance of user level device drivers built using our model. And finally chapter 7 concludes the thesis.

Chapter 2

Related Work

Many techniques have been used to extend the operating system functionality at the user level. In particular, several user level file system implementation techniques have been proposed. For example, the Ufo file system implementation [1] has a catcher process that uses the Solaris */proc* interface to intercept system calls made by an application process. Another simple approach to modify the behavior of system calls is to replace the standard dynamically linked libraries by newer versions. This approach is used by the Jade [9] and Prospero [7] file systems. WebFS [11] and Linux Userfs [2] both implement kernel extensions that allow part of the file system functionality to be implemented at the user level. In both these systems, a loadable kernel module implements a file system that, instead of implementing the file operations itself, simply relays the file requests to a user level server process and returns its reply to the application.

Most operating systems, including Linux [3] and Windows [6] allow device drivers to be implemented as loadable modules. This allows support for new devices to be added to the system at run-time. However the driver code still executes in the kernel mode in this model and therefore the development and testing of a device driver is still very difficult.

Sprite allows user level drivers for pseudo-devices by transparently mapping operations on a pseudo-device into a request-response exchange with a server process [12].

Sprite uses pseudo-devices to implement its terminal drivers, the internet protocol suite and the X-11 window system server at the user level.

Reference [4] describes a proxy driver for Windows NT which relays device access requests to a user level server. Our approach is similar to this and the Sprite approach. However since Linux allows privileged processes to directly access device controllers, we are able to implement user level drivers for physical devices as well.

The WinDriver driver development tool kit from Jungo [5] is a commercial product that allows user level implementation of device drivers for both Windows and Linux. In this system a device driver is written as a library that has to be linked with the application programs. The toolkit has a kernel module that provides low level device access facilities to such a driver. For efficiency, some of the critical driver code can be moved to the kernel. The main drawback of this system is that the application interface for accessing a device with a user level driver is completely different from the usual interface for devices. Also the kernel code cannot interact with a device that has a user level driver. Thus, for instance it is not possible to mount and use a file system residing on a block device that has a user level driver.

Chapter 3

Linux Device Drivers

Under Linux, as in all other Unix variants, devices are classified as character and block devices. Character devices handle data in the form of individual bytes whereas block devices handle data in form of blocks. Each distinct device is identified using a pair of numbers called the major and minor numbers. The major number represents the class or type of the device while the minor number represents the instance of the device within the class. In Linux we can have a total of 256 classes of devices (major number) with each of them having upto 256 instances (minor numbers). All devices with the same major number share the same driver code. As an example *lp0*, *lp1* and *lp2* are three devices representing the three parallel ports on a Linux system. All of them have the same major number 6, and thus share the same driver code (line printer driver), while the minor numbers are 0, 1, and 2 respectively.

Linux provides a file like interface for all devices. Each device is represented by a node in the file system and can be accessed using the usual *open*, *read*, *write*, etc., system calls. In addition, an *ioctl* system call can be used with device files to control the behavior of the device or of the device driver. An ioctl call takes a command and optionally some data as arguments. Each device driver defines its set of ioctl commands specific to the device.

A device driver in Linux has to implement a standard set of interface functions that are called by the kernel file system code. From version 2.4 onwards of the

Linux kernel, this set of interface functions is different for character and block device drivers. A character device driver has to implement functions to open and close the device, read, write, perform ioctl, poll etc. A block device driver does not need to implement functions to read or write the device. Instead it has to implement a request function that handles both read and write requests. In addition a block device driver also has some functions to check for change in media, re-validate the media etc. A detailed description of Linux device drivers can be obtained in reference [10].

Chapter 4

The Userdev Framework

The Userdev framework allows device drivers to be implemented at the user level. Our primary goal for this framework was that the applications should be able to use exactly the same interface for accessing a device with a user level device driver as for accessing devices with the usual kernel resident drivers. This will ensure that existing applications can continue to work without any modifications, re-compilation or even relinking. Both character and block devices should be supported. Also it should be easy to develop user level device drivers so that drivers for new devices can be easily developed.

Figure 4.1 gives an overview of the Userdev framework. It consists of a generic device independent kernel driver called the Userdev driver, and a Userdev library. The Userdev driver implements the usual interface expected of a kernel device driver and thus appears to the kernel as just another device driver. The Userdev driver implements the interface for both character and block devices. In Linux there can be upto 256 minor numbers for a specific major number. Thus the Userdev driver can support user level device drivers for upto 256 block devices and 256 character devices.

Actual device drivers are implemented in the Userdev framework as user level processes. A driver process first registers itself with the Userdev driver specifying a minor number that it will handle. A single process can register for multiple minor

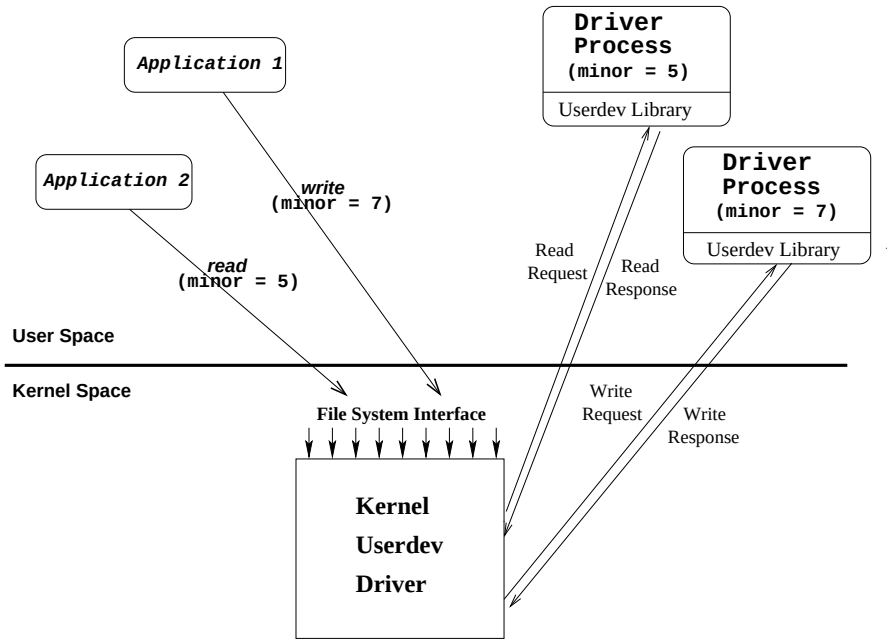


Figure 4.1: System Architecture

numbers. After this, the Userdev driver redirects all requests for any of these minor numbers to the driver process by sending messages on a pipe. The driver process handles the requests and returns the replies to the Userdev driver using another pipe. The Userdev library hides the details of this message passing and exports a simple interface so that driver development is easier.

In Linux, root processes can gain permission to directly access I/O controllers using the *iopl* and the *ioperm* system calls. This facility can be used to build user level device drivers for actual hardware devices. In order to access memory mapped devices, the existing `/dev/mem` device can be used. To avoid the overhead of system calls to use `/dev/mem`, the appropriate region of this device can be mapped to the address space of the driver process using the *mmap* system call.

The Userdev framework has been implemented for the Linux kernel version 2.4. In the following two sections, we describe some implementation details of the Userdev

driver and the Userdev library.

4.1 The Userdev Driver

The Userdev driver is a generic redirection layer that passes requests for access to devices to the user level drivers. The Userdev driver is a driver for both character and block devices and thus implements the kernel interfaces required for both kinds of drivers. The current implementation of the Userdev driver uses major number 248 for both character and block devices. A pair of unnamed pipes is used to communicate with a user level driver process. Minor number 0 is treated as special and *ioctl* calls on this minor number are used for driver registration and other down calls from the user level drivers to the Userdev driver.

Any driver process first needs to register with the kernel Userdev driver before it can obtain requests for access to the device. This operation is called as the *attach* operation. Similarly when the driver process wants to stop accepting requests from the kernel Userdev driver, it can unregister itself. This operation is called as the *detach operation.* The driver process uses specific ioctl calls on minor number 0 of the kernel Userdev driver for both these operations.

The information that the driver process needs to send to the kernel Userdev driver during the attach operation includes: the file descriptors for the kernel driver to read and write from, the name of the device file (thus indicating the minor number) for which the process would accept requests, a list of ioctls that are supported, a function mask indicating the interface functions supported by the driver and finally optional data for specifying device configuration for block devices.

When a driver process calls ioctl on minor number 0 of the Userdev driver to attach to a device, the Userdev driver checks if the minor number for which the driver process wants to accept requests is free. Then it checks that the file descriptor on which this ioctl call is being made is open for writing. That is, write permission on the device file corresponding to minor number 0 of the Userdev driver is required

by a process to act as a user level driver. If these checks are passed, the Userdev driver returns an id, termed as the *Userdev id*, to the driver process to identify the driver process on subsequent ioctl calls to the Userdev driver.

When the driver process exits, it can detach itself from the Userdev driver again using an ioctl call on minor 0. A check is made by the Userdev driver to make sure that the detach call is from the same process that earlier attached to the device. The implementation also automatically detaches a driver process that has terminated without detaching itself.

Once a user level driver has attached itself to a minor number, any call to the Userdev driver from the kernel for that minor number is assembled into a request message and is sent to the driver process using the file descriptor specified at attach time. The process that made the call sleeps in the kernel. When the reply is received (on the second pipe), this process wakes up and returns from its call to the Userdev driver.

Multiple processes can simultaneously try to access the same device. In this case, the Userdev driver can send another request to the driver process without waiting for the responses to the previous ones. The driver process is also permitted to send replies in an order different from the order in which the requests were received. This is done so that fast operations do not necessarily have to wait for previously initiated slower operations to complete. In order that requests and replies can be correlated with each other, each request message carries a unique request id and the corresponding reply must also carry the same request id.

Each request and reply message consists of a fixed length header followed by a variable sized data portion. The header contains the operation code, the request id, and the size of the data part. The contents of the data part vary depending on the operation. The list of messages and message formats can be found in Appendix A.

The poll and fasync interface functions of the Userdev driver have to be treated differently since these functions are expected to wakeup or signal respectively the calling process whenever input or output is possible. To implement this functionality, the driver process has to make an ioctl call on minor number 0 whenever I/O is possible and a poll or fasync request was previously received by it.

4.1.1 Support for Interrupts

Many I/O devices are interrupt driven. To enable implementation of drivers for such devices at the user level, a mechanism to communicate interrupt occurrence events to a user level process is required. Since Unix signals behave very much like interrupts, it is natural to use signals for this communication. In the Userdev framework, a driver process can request access to a certain interrupt by making an ioctl call on minor 0 of the Userdev driver specifying the interrupt number and the number of the signal that should be delivered to the process when the interrupt occurs. The Userdev driver then installs its own interrupt handler for this interrupt. This interrupt handler simply delivers the specified signal to the driver process. The driver process can install a signal handler for this signal and do the processing required for servicing the interrupt in the signal handler.

4.1.2 Support for Direct Memory Access:

Direct Memory Access (DMA) is used to speed up bulk data transfers between devices and memory. Even though it is possible in Linux for a user level process to directly program the DMA controller, this should not really be done since kernel drivers for other devices may also use the DMA controller. Further some DMA controllers can access only a limited range of the physical memory. For these reasons, the DMA controller should only be programmed by the kernel code. To make the DMA facility available to the user level device drivers, the Userdev driver implements certain ioctl calls on minor number 0. These ioctl calls can be used by a driver process to request for a given DMA channel, initiate a DMA operation, enquire the status of an on-going DMA operation and free the DMA channel. The Userdev

driver allocates a DMA buffer when a DMA operation is initiated. This buffer is used for the data transfer. In case of a read operation an additional ioctl on minor 0 is needed to copy the data read from the device back to the driver process. The driver process can get to know when to issue this ioctl call either through an interrupt from the device or by enquiring the status of the DMA operation.

4.1.3 The Signal Message

If a process that is blocked while accessing a device receives a signal, typically the operation is aborted and the process returns from the driver code with an error. To achieve similar behavior in case of user level device drivers, the Userdev driver sends a *signal* message to the driver process when a process waiting for reply to a request receives a signal. The signal message essentially requests the driver process to abort an ongoing operation. The request id corresponding to the operation to be aborted is included as data in the signal message. The driver process, on receiving this message, will usually abort the specified operation and send a reply for the aborted request indicating an error condition. The signal message itself has no reply.

4.2 The Userdev Library

The Userdev library forms the other part of the framework for writing user level device drivers. It hides the details of the protocol to be followed with the kernel Userdev driver and thus reduces the effort in constructing a user level device driver. The library not only takes care of the job of obtaining requests from the Userdev driver but also provides wrapper functions for the user level driver to interact with the kernel Userdev driver.

A user level device driver written using the Userdev library needs to implement a set of interface functions that is similar to the set of functions that a kernel driver implements. The *userdev_start* function of the library implements the main loop of the driver. It essentially waits continuously for requests from the kernel and on receiving a request message calls the appropriate function in the driver

code. This function is required to handle the request and send the response message using the *userdev_send_response* function of the library. We decided to require the driver code to explicitly send the response message since this allows the driver to delay sending the response. Most drivers are expected to be multi-threaded so that multiple requests can be handled concurrently.

The Userdev library also provides high level functions corresponding to the ioctl calls available on minor number 0 of the Userdev driver. This includes functions to attach to and detach from a minor number, to request notification of interrupts, DMA related functions etc. The library also provides utility functions that many drivers require. These include a microsecond delay loop, implementation of timer queue, functions to access I/O ports directly etc. The complete library interface is given in Appendix B.

Chapter 5

Prototype Drivers

To demonstrate the feasibility of using the Userdev framework to develop user level device drivers, we have developed three prototype drivers. These are drivers for a parallel port printer, floppy disk drive, and RAM disk. The Linux kernel already has drivers for all these three devices and we actually used the existing kernel code for implementing the parallel port printer and the floppy disk drivers in the Userdev framework. In this section we briefly describe the implementation of these three user level device drivers.

5.1 Parallel Port Printer Driver

The kernel driver for the parallel port printer has a three layer architecture consisting of the *lp driver* module, the *parport* module and the *parport_pc module*. The lp driver module implements the device driver interface while the actual work is done by parport and parport_pc modules. The parport module is a generic unit that multiplexes various kernel drivers over low-level drivers which access the parallel port hardware interface. The upper layer drivers, such as the lp driver, register with the parport module that provides them the interface for accessing the device while the actual operation is done by the low-level device specific drivers, such as the parport_pc module, that register with the parport module. This architecture allows

multiple drivers to operate on a single device simultaneously with the contention being resolved using a resource claim-release mechanism followed by the upper layer drivers to avoid interference with one another.

The user level implementation of the parallel port printer driver has a similar architecture. Here the parport and parport_pc modules are libraries. This allows easy development of the drivers for other parallel port devices such as scanners.

Since the line printer is not usually accessed simultaneously by multiple processes, the driver handles only one request at a time. It does not allow more than one process to have the device open and thus enforces mutual exclusion. Similar to the kernel implementation, our printer driver can use either polling or interrupt driven I/O, based on a command line option. In the interrupt mode, if the driver detects missed interrupts it automatically switches to polling mode.

The write operation of the printer driver is the only one that can block (in the interrupt mode) or take a long time to complete (in polling mode). In order to abort the write operation if a signal message is received, the write operation is executed by an independent thread while the main thread goes back to waiting for requests. All other operations complete quickly without blocking and are therefore executed by the main thread itself.

5.2 RAM Disk Driver

A RAM disk driver is the simplest block device driver as the only operation involved is copying bytes across the memory. A portion of the main memory allocated by the driver is treated as a block device and operations are performed on it. The user level RAM disk driver built using the Userdev framework has dummy functions for most of the driver interface functions, that is they just return success, except for ioctl and request functions. The ioctl function handles HDIO_GETGEO and BLKGETSIZE ioctl commands to return a fake disk geometry and device size respectively. These are mandatory ioctls for any block device. The request function

translates the sector number present in the block device I/O request into a memory address from which the required number of bytes are copied to the transfer address given in the request or vice-versa depending on whether the request is for reading or writing data.

5.3 Floppy Disk Driver

The floppy disk driver is a complex multi-threaded block device driver. The kernel floppy disk driver extensively uses kernel support utilities such as task queues and kernel timers to delegate work to another thread, and to delay an operation for a certain period of time respectively. The driver not only serves read and write requests from the file system buffer cache, but also serves ioctl requests to change the floppy drive parameters, to get or set the disk geometry, format a given track, etc.

The user level floppy disk driver has been implemented by porting the kernel floppy disk driver to the user space. The driver is written as a multi-threaded application using the *pthread* user level thread package [8]. The kernel task queues and timers used by the kernel driver are replaced by the corresponding user level libraries. In order to implement the locking mechanism for serializing the access to the floppy drive controller, thread synchronization mechanisms such as mutexes and condition variables are used. Interrupts from the floppy drive controller are delegated to the user level driver using the Userdev framework. All the driver interface functions except release, can potentially block, so a separate thread is created for executing each of them.

Chapter 6

Performance

In this section we describe the experiments conducted to measure the performance of the prototype drivers and the Userdev framework. For the floppy drive and the RAM disk drivers, the performance was compared with that of existing kernel drivers for these devices. In addition, we performed an experiment to measure the maximum interrupt rate that can be handled by a user level device driver. All experiments were performed on a PC with a 233 MHz Pentium processor and 128 MB RAM. The following sections describe the experiments and their results.

6.1 Performance of Floppy and RAM Disk Drivers

For both floppy disk and RAM disk drivers we measured the time taken to mount a file system from the device, and the read and write data rates. These figures were also measured for the existing kernel drivers for the floppy drive and the RAM disk. The results are shown in Table 6.1 and Table 6.2 respectively. The results show that the data rates of the user level floppy driver are only slightly lower than those of the corresponding kernel driver. This is expected since the dominating factor in read and write time is the device delay. The user level RAM disk driver on the other hand performs significantly worse than the kernel RAM disk driver. Again this is expected since in this case there is no physical device involved and the penalty of extra data movement and context switching in case of the user level driver

is significant. However the RAM disk driver performance is really the worst case. Most physical devices are orders of magnitude slower than the CPU and therefore the penalties associated with a user level driver would be significantly lower in relative terms for these devices. Surprisingly the mount time for the user level floppy driver is *lower* than that for the kernel driver. This is a repeatable observation but currently we cannot explain it.

Parameter	Kernel Space Driver	User Space Driver
Mount Time	1.75 sec	1.57 sec
Read Data Rate	5.74 KB/s	5.59 KB/s
Write Data Rate	7.46 KB/s	7.25 KB/s

Table 6.1: Performance comparison of user level and kernel level floppy disk drivers. A FAT file system was used to measure the mount time.

Parameter	Kernel Space Driver	User Space Driver
Mount Time	1 msec	2.2 msec
Read Data Rate	2.57 MB/s	1.68 MB/s
Write Data Rate	2.57 MB/s	1.58 MB/s

Table 6.2: Performance comparison of user level and kernel level RAM disk drivers. An Ext2 file system was used to measure the mount time.

6.2 Interrupt Rate

To measure the maximum interrupt rate that can be handled by a user level driver, we short-circuited pins 9 and 10 of the parallel port interface. Pin 9 is the most significant data bit and pin 10 is the acknowledgement bit. The parallel port controller raises an interrupt whenever the acknowledgement bit falls to 0. We then ran a process that repeatedly wrote a byte of data to the parallel port with the most significant bit set to 0. The process directly accessed the parallel port controller to do this and its rate of writing could be controlled. Thus we were able to generate

interrupts at any desired rate. We ran a dummy user level parallel port driver whose interrupt handler simply incremented the number of interrupts received and executed a delay loop to simulate interrupt processing.

Experiments showed that with a 20μsec interrupt handling time, the dummy driver was able to handle about 11,000 interrupts per second. This is greater than the actual interrupt rate from most devices. For example, data transfers in units of 4 KB from a fully busy Ultra-2 SCSI disk with a bandwidth of 80 MBps and 50% bandwidth utilization, would result in about 10,000 interrupts per second.

Chapter 7

Conclusions

In this thesis, we have described the design and implementation of the Userdev framework that allows device drivers to be implemented at the user level in Linux. We have also implemented user level drivers for the parallel port printer, the floppy drive and the RAM disk. Experiments show that the performance of user level drivers is only slightly worse than that of kernel level drivers. For performance critical devices, the Userdev framework can be used to quickly develop and test a user level driver which can later be ported to the kernel. For slow devices, a user level driver may be all that is ever required. The Userdev framework also allows support for remote devices or "pseudo-devices" for which modifying the kernel may not be appropriate.

Our future plans include gaining more experience with the Userdev framework by writing more user level device drivers, including drivers to transparently access remote terminals and scanners.

The current implementation of the Userdev framework and the prototype drivers can be downloaded from http://www.cse.iitk.ac.in/users/deepak/userdev

Appendix A

List of Userdev Protocol Messages

The protocol messages that are exchanged between the Userdev driver and the driver process consist of a fixed length header field and a variable length data field. The header field consists of:

- operation code: denoting the operation in question
- request id: shared by request and response messages for co-relation
- size of the data field to follow

The contents of the data field for various operations is given in the table below:

Operation	Request Packet	Response Packet
read	- amount to be read - operation flags - position on device	- number of bytes read - data read from device - new position on device
write	- amount to be written - position on device - data to be written - operation flags	- number of bytes written - new position on device
poll		- poll operation result
ioctl	- ioctl command - ioctl data (SET) - size of data	- result of operation - ioctl data (GET) - size of data
open	- file open flags - mode of operation	- result of operation
flush		- result of operation
close		- result of operation
fsync	- datasync flag	- result of operation
fasync		- result of operation
check_media_change		- result of operation
revalidate		- result of operation
mediactl		- result of operation
readv	same as read	same as read
writev	same as write	same as write
signal message	- type of message - request id of pending request - optional data - size of data	
request function	- command code - sector number - transfer length - cluster size - data being sent for write - size of data being sent	- result of operation - size of data received - data received for read

Table A.1: List of messages and their format

Appendix B

Userdev Library Interface

The interface between the Userdev library and the user level device driver consists of a set of functions known as *Userdev operations* that the device driver can implement to serve requests from the Userdev kernel driver. The library also provides functions for interacting with the Userdev kernel driver and other utility routines that many drivers require.

B.1 Userdev Operations

Userdev operations forms the set of functions of the user level device driver that can be called by the Userdev library to serve requests from the Userdev kernel driver. It is similar to the *file operations* structure of the kernel, used for communicating the list of functions implemented by the kernel device driver to the rest of the kernel. When the library obtains a packet from the Userdev kernel driver, it looks into the Userdev operations set to check if the driver has implemented the corresponding function, in which case the function is called passing the constituents of the packet.

The Userdev operations is represented by a structure consists of the pointers to functions corresponding to each operation the Userdev kernel driver can request, as shown below. The driver fills in each field of the structure with the name of the function serving the request or NULL if the function has not implemented and

communicates the structure to the library along with the call to *userdev_attach* library function.

```
struct userdev_operations
{
        int devtype;
        void (*read)(int id, unsigned int size, int flags, long long off,
             int reqid);
        void (*write)(int id, int len, long long off, char* data,
             int flags, int reqid);
        void (*poll)(int id, int reqid);
        void (*ioctl)(int id, int command, void* data, int size,
             int reqid);
        void (*open)(int id, unsigned int flags, mode_t mode,
             int reqid);
        void (*flush)(int id, int reqid);
        void (*close)(int id, int reqid);
        void (*fsync)(int id, int datasync, int reqid);
        void (*fasync)(int id, int reqid);
        void (*check_media_change)(int id, int reqid);
        void (*revalidate)(int id, int reqid);
        void (*mediactl)(int id, int op, int optarg, int reqid);
        void (*message)(int id, int type, int reqid, void *data,
             int size);
        void (*request)(int id, int command, long sector, int length,
             int clustersize, void* data, int datasize, int reqid);
};
```

The arguments to the functions in the above structure correspond to the contents of the data field of the corresponding request message, in addition to the Userdev id *(id)* and request id *(reqid)*.

B.2 Library Routines

The various routines provided by the library to interact with the Userdev kernel driver are:

B.2.1 userdev_attach

This function is called to attach a user level driver process to the Userdev kernel driver. It takes the device filename to attach to, an array of ioctl data elements specifying the list of allowed ioctl numbers and maximum length of data handled by each, the number of ioctl data elements, a pointer to the driver's Userdev operations structure and an optional data parameter used for block devices. The Userdev operations structure is stored in the library's internal data structure, maintained for each attached driver process, for use by other routines of the library. The function returns the Userdev id of the driver process in case the attach operation had been successful and an error number in case an error was encountered.

B.2.2 userdev_detach

This function is called to detach the driver process from Userdev kernel driver, identifying it using the Userdev id given as the function argument. The result of the detach operation is returned.

B.2.3 userdev_inform_poll_in

This function is called to ask the Userdev kernel driver to inform the availability of data to anybody polling the device for input data. The Userdev id, passed as the function argument is used to identify the attached driver process. The value returned by the function is the result of the operation.

B.2.4 userdev_inform_poll_out

This function is called to ask the Userdev kernel driver to inform the availability of the device for output to anybody polling the device for output. The Userdev id

passed as the function argument is used to identify the attached driver process. The result of the operation is the return value of the function.

B.2.5 userdev_inform_fasync_io

This function is called to ask the Userdev kernel driver to inform the availability of data to anyone using asynchronous I/O and waiting for some I/O activity to take place. The Userdev id is passed as the function for identifying the attached driver process and the result of the operation is returned back to the caller of the function.

B.2.6 userdev_request_irq

This function is called to request the Userdev kernel driver to inform about the occurrence of interrupts on a given IRQ line using a signal. The arguments to the function are the IRQ line number, the signal number corresponding to it, pointer to the signal handler function and lastly the Userdev id of the attached driver process. The result of the operation is returned back to the caller.

B.2.7 userdev_free_irq

This function is used to ask the Userdev kernel driver to stop signalling the occurrence of an interrupt. The result of the operation is returned back to the caller. The function also resets the signal disposition of the signal corresponding to an interrupt on the IRQ line to the default value.

B.2.8 userdev_request_dma

This function is used to request for a DMA channel from the kernel. The arguments of the function are the DMA channel number and the Userdev id of the driver process. The result of the operation is returned to the caller.

B.2.9 userdev_free_dma

This function is used to free the DMA channel acquired by the driver process to the kernel resource pool. The argument passed to the function is the Userdev id of the driver process. The result of the operation is returned to the caller.

B.2.10 userdev_start_dma

This function is used to start a DMA operation on the DMA channel acquired by the driver process. The arguments to the function are the mode of the DMA operation, the data buffer, the data transfer length and of course the Userdev id to identify the driver process. The return value of the function is the result of the operation.

B.2.11 userdev_check_dma

This function is used to obtain the status of an ongoing DMA operation. The Userdev id is passed as the function argument to identify the driver process and the result of the operation, that is the number of bytes remaining in the transfer is returned to the caller.

B.2.12 userdev_copy_dma

This function is used to copy the contents of the kernel DMA buffer into the buffer passed as the function argument. The Userdev id of the driver process forms the other argument of the function. The function returns the result of the copy operation obtained from the kernel.

B.2.13 userdev_enable_dma

This function enables a DMA operation incase one has been suspended. The Userdev id is passed as the function argument to identify the driver process sending the request. The return value of the function is the result of the operation returned by the kernel.

B.2.14 userdev_disable_dma

This function disables a DMA operation incase one is in progress. The Userdev id is passed as the function argument to identify the driver process sending the request. The return value of the function is the result of the operation returned by the kernel.

B.2.15 userdev_start

This function is the work horse function of the library and is called once all initializations have been completed and the driver process is ready to serve requests from the Userdev kernel driver. This function does not return to the caller unless an error is encountered. It continuously obtains requests from the Userdev kernel driver and calls the corresponding function of the Userdev operations structure specified during the call to the *userdev_attach* function.

B.2.16 userdev_send_response

This function is used to send the response to a request, obtained from the Userdev kernel driver, by the driver function that was called to serve the request. The arguments of the function are the Userdev id of the driver process, the request id of the request, the operation code, the result of the operation, the size of data to be sent in the response and the data itself. The function has no return value.

B.2.17 Utility functions

The various utility functions that are provided by the Userdev library can be classified into following categories:

- *Block Device Request Queue*

Simulates the request queue mechanism for block devices where in a request queue can be defined and block device requests can be enqueued and dequeued from it. The functions available are:

1. *userdev_ add_ request:* function to add a request to the request queue given the userdev id, command, starting sector number, transfer length, cluster size, data to be transferred in case of read operation, the size of the data and the request id. The function returns a pointer to a *userdev_ blk_ request* structure added to the request queue on success and NULL otherwise.

2. *userdev_ end_ request:* function to remove a request from the front of the request queue and to signal any threads waiting on the condition variable of the request. The function argument is dummy and is used to keep the signature similar to the kernel function.

3. *userdev_ blk_ requestq_ cleanup:* function to remove all the requests in the request queue.

Signal List

Defines a list to store the signal messages that have arrived for various pending requests and provides functions to add a message to the list, remove a message from it and of course check if a signal message has arrived for a given request (identified by the request id). The functions available are:

1. *userdev_ add_ sig_ list:* function to add an item with the given request id to the signal list

2. *userdev_ del_ sig_ list:* function to remove all items in the signal list for the given request id

3. *userdev_ check_ sig_ list:* function to check if a signal message has arrived for the given request id. The function returns 1 in case an item is found and 0 otherwise.

Task Queue

Simulates the task queue mechanism of the kernel wherein a task queue can be defined and tasks can be added to it which are executed by a task thread that runs in parallel and repeatedly scans the task queue. The functions available are:

1. *userdev_queue_task:* function to add the given task to the given task queue.

2. *userdev_run_task_queue:* function to create the task thread for the given task queue and assign the given function as the initialization function of the task thread.

3. *userdev_task_queue_cleanup:* function to set the stop flag of the given task queue in order to stop the task thread from polling the task queue.

Timer Queue

Simulates the kernel timer mechanism wherein a task can be added to the timer queue specifying the time in jiffies at which it must be executed. A thread runs through the timer queue which is sorted by task expiry time, and executes the tasks as and when their timer expires. The functions available are:

1. *userdev_add_timer:* function to add the given timer task to the timer queue while keeping the queue sorted by the expiry times of the tasks.

2. *userdev_del_timer:* function to remove the given timer task from the timer queue. The function returns 1 if the task was found in the timer queue and 0 otherwise.

3. *userdev_mod_timer:* function to modify the expiry time of the given timer task if it is present in the timer queue. The function returns 1 if the expiry time of the timer task has been modified and 0 otherwise.

4. *userdev_timer_pending:* function to check if the given timer task is still to be executed that is, it is still present in the timer queue. The function returns 1 if the timer task has been found and 0 otherwise.

5. *userdev_get_jiffies:* function that returns the current time in 100th of a second since epoch. This function is used for specifying the expiry time of the timer tasks.

6. *userdev_ timer_ queue_ cleanup:* function to set the *timerq_ stop* flag inorder to stop the timer queue thread.

7. *userdev_ init_ timer_ queue:* function to initialize the timer queue, create a timer queue thread, install a signal handler for the alarm signal used to keep track of time when executing the tasks. The function argument is a pointer to the initialization function to be executed when the timer thread begins.

Microsecond Delay Loop

Provides the facility of busy waiting for a certain number of micro seconds. A loop calibrated for number of iterations required to get a microsecond delay is used to produce the required delay. The functions available are:

1. *userdev_ calibrate_ delay:* function to calibrate a loop to find the number of iterations required for a delay of 1 microsecond. This function needs to be called before the *userdev_ udelay* function can be used. The value of *loops_ per_ usec* cannot be determined ahead as it is dependent on the speed of operation of the processor, the load on the system etc.

2. *userdev_ udelay:* function to be called to obtain a delay of given number of microseconds.

I/O Address space

Simulates the kernel mechanism of registering I/O addresses so that two device drivers do not simultaneously use a given address. The registration is recorded in a file so that it is effective across process boundaries and can prevent two driver processes from writing to the same I/O address space. The functions available are:

1. *userdev_ check_ ioregion:* function to check if the I/O address space, specified by the given starting address and number of addresses is free. The function returns 0 if the region is free and negative number otherwise.

2. *userdev_request_ioregion:* function to reserve the I/O address space, specified by the given starting address and number of addresses under the given device name. The function returns 0 on success and a negative number otherwise.

3. *userdev_release_ioregion:* function to release the I/O address space, specified by the given starting address and number of addresses. The function returns 0 on success and a negative number otherwise.

I/O port access

Provides functions for setting and resetting permissions on the given I/O port and also wrapper functions for each of the port access functions. The functions available are:

1. *userdev_set_ioperm:* function to set the I/O permission for the current process on the given port number. The function returns 0 on success and a negative error number otherwise.

2. *userdev_reset_ioperm:* function to reset the I/O permission for the current process on the given port number. The function returns 0 on success and a negative error number otherwise.

3. *userdev_inportb:* function to read a byte from the specified port. The value read is returned by the function.

4. *userdev_outportb:* function to write a byte to the specified port number.

5. *userdev_inportw:* function to read a word from the specified port. The value read is returned by the function.

6. *userdev_outportw:* function to write a word to the specified port number.

7. *userdev_inportd:* function to read a double word from the specified port. The value read is returned by the function.

8. *userdev_outportd:* function to write a double word to the specified port number.

9. *userdev_reset_all_perms:* function to reset all the I/O permissions acquired by the current process.

Bibliography

[1] Alexandrov, A. D., Ibel, M., Schauser, K. E., and Scheiman, C. J. Extending the operating system at the user-level: the ufo global filesystem. In *USENIX Annual Technical Conference* (Anaheim, CA, 1997), pp. 77–90.

[2] Fitzhardinge, J. Userfs: A user file system for linux. ftp://sunsite.unc.edu/pub/Linux/ALPHA/userfs, 1997.

[3] Henderson, B. Linux loadable kernel module how to. http://www.tldp.org/HOWTO/Module-HOWTO/index.html, August 2001.

[4] Hunt, G. C. Creating user-mode device drivers with a proxy. In *1st USENIX Windows NT Workshop* (Seattle,WA, August 1997), pp. 55–59.

[5] Jungo. Windriver for linux. http://www.jungo.com/linux.html.

[6] Microsoft. The driver development kit. http://www.microsoft.com/ddk/.

[7] Neuman, B. C. The prospero file system: A global system based on the virtual system model. In *Computing Systems* (Fall 1992), pp. 5(4):407–432.

[8] Nichols, B., Buttlar, D., and Farrell, J. P. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, 1 ed. O'Reilly & Associates, Inc, September 1996. http://www.oreilly.com/catalog/pthread/.

[9] Rao, H. C., and Peterson, L. L. Accessing files in an internet: the jade file system. In *IEEE Transactions on Software Engineering* (June 1993), pp. 19(6):613–624.

[10] Rubini, A., and Corbet, J. *Linux Device Drivers*, 2 ed. O'Reilly & Associates, Inc, June 2001. http://linux.interpuntonet.it/doc/linuxdriver2/index.html.

[11] Vahdat, A., Eastham, P., Yoshokawa, C., Belani, E., Anderson, T., D.Culler, and Dahlin, M. Webos: Operating system services for wide area applications. Tech. Rep. CSD-97-938, Dept of EECS, U. C. Berkeley, June 1997.

[12] Welch, B. B., and Ousterhoust, J. Pseudo-devices: User-level extensions to the sprite file system. In *Summer USENIX Conference* (June 1988).