# Creating virtual "soft" devices with User-mode Linux

Sapan Bhatia  Laurent Réveillère

LaBRI/INRIA
ENSEIRB
1 Ave. du Dr. Albert Schweitzer
Domaine Universitaire
33400 Talence, FRANCE
{bhatia,reveillere}@labri.fr

**Abstract**

Developing device drivers can be highly tedious as it entails direct interaction with hardware devices, which are difficult to analyse in trying to find the cause of unexpected behaviour. User-mode Linux (UML) [3, 4] simplifies this task by allowing developers to test and debug their device drivers in user-space. In this paper, we describe a systematic approach to create virtual *soft* devices using UML for the purpose of testing device drivers while they are still in the stages of development. Soft devices run as user-space processes and can have a GUI interface. We have used the existing emulation capabilities of UML and extended them by adding some of our own. We have designed a language named *Saint* to specify soft devices, and implemented a virtual coffee-machine soft device as a proof of concept.

## 1 Introduction

Device drivers are a key component of operating systems. They insulate application developers from the details of underlying hardware, thus implementing the mechanism of the software interface with the device. The use of this mechanism, or the actual *policies* are left to application developers. Writing device drivers is a highly rigorous task especially when considering *hardware operating code* (*i.e.* code interacting with hardware). This layer of code is low level and known to be *error prone*. This is aggravated by the fact that minor errors can be expected to have disastrous effects on the system, as hardware devices are not robust to erroneous code.

It is thus imperative that device drivers be thoroughly tested in a mock test environment before they are actually made to interact with real hardware. User-mode Linux (UML) [3, 4] provides such an environment by running the Linux kernel in user space. This means that device drivers are executed in user context, reducing the penalty of faulty code. Instead of system crashes, faulty device drivers cause exceptions in the UML kernel, which causes the UML processes to terminate. Device driver developers can also make use of user-space libraries and applications for extensive logging and debugging.

UML can also be made to emulate I/O memory and interrupts. Files on the host system can be mapped into UML as I/O memory. Similarly, `SIGIO` signals can be used to generate interrupts inside UML. This suggests that one should be able to emulate a complete device, and use the emulated device to test and experiment with a device driver. One of the things that UML lacked in this respect was the ability to emulate I/O port memory. We have implemented I/O port emulation for UML, which is to be merged into the UML tree in the mainline Linux kernel. Apart from this, we modified the timer modules for more precise timing, as device drivers often rely on it, and the interrupts sub-system to be able to pass a file to UML so that device drivers could transparently register IRQ handlers. This file would be used to generate interrupts inside UML.

In this paper, we show how one can use these emulation capabilities of UML to create virtual soft devices to be used to test device drivers. Such devices run as user-space processes and can have GUI front ends to display register states and give information about the operation of the device interactively. We also envisage this concept to be used to profile actual devices. A software implementation of a real hardware device can be used to analyse possible settings and configurations by subjecting the device to real application loads. *I.e.* processes in UML could use the device oblivious of the fact that it is emulated, and the process emulating the device could gather valuable information from the work load. To simplify the creation of soft devices, we have designed a language named *Saint* to specify them. Saint is a *Domain Specific Language (DSL)* [**?**] that generates stubs used in the device implementation.

To illustrate the concept of soft devices, we have implemented a conceptual device, a coffee machine, as a soft device and written a device driver to control it. The coffee machine application runs on the host machine and has a GUI interface. Within UML, the coffee machine can be manipulated using port memory and hardware interrupts. This device has already been used in a senior year device drivers course.

In Section 2, we give a brief overview of User-mode Linux. In Section 3, we outline how we have emulated hardware resources to create soft devices. In Section 4, we discuss the possible uses of soft devices. In Section 5, we describe our coffee machine device, which we use as an example while outlining the procedure to create and use soft devices. In Section 6, we detail the process of creating and using a soft device and in Section 7 we discuss how device drivers access it from UML. Finally, in Section 8, we describe Saint, a language we have designed to specify soft devices and simplify the process of their development.

## 2 User-mode Linux

User-mode Linux is a port of the Linux kernel that runs on Linux in a set of processes. The result is a user-space virtual machine using simulated hardware, constructed from services provided by a Linux kernel running on hardware (a *host kernel*). A Linux virtual machine is capable of running all of the applications and services available on the host architecture. Instead of a hardware architecture, as is usual, UML uses the system call interface of the Linux kernel as its underlying hardware interface. The code that implements this is under the *arch* interface of the Linux kernel. The rest of the kernel is an unmodified, full-fledged Linux kernel with all its features including scheduling, memory management and dynamically loaded modules.

UML, like the actual kernel, distinguishes between a privileged *kernel mode* in which only trusted code runs and a non-privileged *user mode* in which operations are arbitrated before

being executed. This is done using the `ptrace` interface to intercept system calls. Thus, system calls issued in kernel context are executed directly without any arbitration, while system calls issued in user context are intercepted by the UML kernel, which annuls them on the host kernel and emulates them. Virtual memory is implemented by treating a file on the host system as a pool of physical memory. The virtual memory system in the Linux kernel then uses this pool of physical memory as usual. When a page needs to be mapped into virtual memory, a corresponding "physical page" in the file is mapped into an appropriate spot in the virtual memory in UML. UML implements I/O device interrupts with `SIGIO`, page faults using `SIGSEGV` and timer interrupts using `SIGALRM`.

# 3 Emulation of hardware resources

In this section, we discuss the emulation of hardware resources, which is used extensively in soft devices. Of the resources that are emulated, I/O memory emulation was fully implemented when we started this project, interrupt emulation needed modification and I/O port emulation needed to be implemented completely.

## 3.1 I/O memory emulation

I/O memory is the main mechanism used by device drivers to communicate with devices. UML has support for I/O memory emulation. This means that a host file can be specified as an I/O region in the UML kernel, so that drivers can locate it and use it as desired, just as they would use physical memory. If UML is executed with the argument `iomem=softiomem,/tmp/softiomem`, then a driver can locate the corresponding memory area in UML using the `find_iomem` routine as follows.

```
device_memory = (void *) find_iomem("softiomem", &iomem_size);
```

This is used by soft devices to make their device memory and memory mapped registers accessible to the device driver.

## 3.2 Interrupt Emulation

In UML, interrupts are emulated using signals and in particular, the `SIGIO` signal. Every file descriptor corresponds to a different IRQ. Interrupt handling in UML is a bit more tedious than on a host kernel. Other than requesting an IRQ and handling interrupts, drivers must also manage the files which will be used to deliver signals. They must also disable the corresponding file descriptor while the interrupt is being handled and re-enable it once it has been handled. This is undesirable when device drivers are being tested, as they are intended to run on the host kernel, and should run under UML as is. To make this possible, we made this process transparent by factoring out the code that handles files and enables and disables file descriptors. For the latter, we used the `irq_desc_t` interface, which is invoked at various stages of handling an interrupt.

## 3.3 I/O port emulation

We had to implement I/O port emulation since it did not exist in UML to begin with. Under Linux, processes gain access to I/O ports by using the `iopl` and `ioperm` system calls. Every

time an I/O request is executed by a process without having done so, an exception is generated by the processor. This exception is relayed to the offending process as a `SIGSEGV` signal with information such as the fault address, the fault type *etc.*. We intercept such exceptions, nullify them and go on to interpret the faulting instruction *i.e.* the I/O call. We apply the result of the instruction to an area of memory we share with the soft device. The soft device is then sent a `SIGUSR1` signal to trigger associated side effects. Finally, we implemented the `iopl` and `ioperm` system calls to manage access permissions to ports in UML.

## 4 Usage scenarios for soft devices

Although so far, we have only used our work for teaching a device drivers course in which students were expected to write a device driver for a soft device, we envisage many diverse uses of soft devices:

- *Making device drivers more robust.* By implementing the behaviour of a device, fully or partially, one can test device drivers in a convenient way. The coffee machine device we have implemented illustrates this. Furthermore, since processes are far more flexible than actual devices, they can be manipulated to create test cases for device drivers which would not be possible to create with real devices. Sometimes, unfavorable conditions may be created by implementing deviant or inconsistent behaviour in the soft device. Unified device drivers for a range of models of a particular device can also be tested in this way, by configuring the soft device to behave specific to particular models.

- *Teaching device drivers and operating system concepts.* Since soft devices are processes that can be restarted when they crash, and made to display an error condition instead of indicating it with a total system freeze, they are especially well suited to the purpose of teaching how to write device drivers.

- *Profiling hardware devices.* Since soft devices can be subject to processes run inside UML, they can be implemented in a way to collect critical information from real-world situations. This goes much beyond simulations, as it makes use of real-world work loads. In an extreme case, it might actually be possible to implement a device in circuit detail so as to record information pertaining to hardware modules and logic elements.

## 5 The coffee machine device

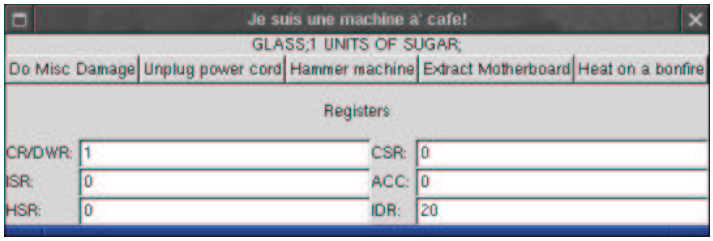


Figure 1: The coffee machine

We have implemented a conceptual device, a coffee machine as a soft device. On the host, the device is a process with a GUI interface (as show in figure 1. Inside UML, this device is accessed using five registers, which can be used to invoke different operations. These are illustrated in Table 1.

| Address | RD | WD |
|---|---|---|
| 0H | Data Waiting Register (DWR) | Command Register (CR) |
| 1H | Interrupt Status Register (ISR) | Reserved |
| 2H | Hardware Status Register (HSR) | Reserved |
| 3H | Container Status Register (CSR) | Reserved |
| 4H | Accumulator (ACC) | Reserved |
| 5H | Reserved | Indexed Data Register (IDR) |

Table 1: Direct Registers

As an example to show how the registers are used, the *command register* is described below.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| AS2 | AS1 | AS0 | PS1 | PS0 | IAD2 | IAD1 | IAD0 |

| Bit | Symbol | Description |
|---|---|---|
| D2 - D0 | IAD0 - IAD2 | **Index Address:** These three bits select which indirect register is to be accessed through the *Indexed Data Register* (IDR). |
| D3 - D4 | PS0 - PS1 | **Push Select:** These two encoded bits control *Push* operations.<br>PS1 PS0<br>0 0 No Operation<br>0 1 Push a glass<br>1 0 Push a stirrer<br>1 1 Push items |
| D5 - D7 | AS0 - AS2 | **Action Select:** These three encoded bits control actions other than a push.<br>AS2 AS1 AS0<br>0 0 0 No Operation<br>0 0 1 Eject the glass<br>0 1 0 Reset the machine<br>0 1 1 State Request<br>1 0 0 Lock<br>1 0 1 Unlock<br>1 1 0 Data Reading Acknowledgment<br>1 1 1 Reserved |

The coffee machine also generates interrupts on certain events, such as error conditions and during state queries, to indicate that new data is available. The type of the interrupt is specified by the *interrupt status register*:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| DW | — | — | — | CNE | OVW | HWF | OPC |

| Bit | Symbol | Description |
|---|---|---|
| D7 | DW | **Data Waiting:** This bit is set when a byte is available in the Accumulator register (`ACC`). This bit is low when no more bytes are available through the Accumulator register. |
| D6 | Reserved | Reserved |
| D5 | Reserved | Reserved |
| D4 | Reserved | Reserved |
| D3 | CNE | **Container Empty:** Indicates that at least one container is empty. The Container Status Register (`CSR`) records status of the various containers. This bit is cleared by the coffee machine itself. |
| D2 | OVW | **Overflow:** Set when no more items can be accomadated in the order. I.e., the dispensing compartment is full. |
| D1 | HWF | **Hardware Failure:** Indicates a hardware failure. The Hardware Status Register (`HSR`) allows one to identify the cause of the failure. This bit is cleared by the coffee machine itself. |
| D0 | OPC | **Operation Completed:** This bit is set by the coffee machine when an operation is completed. This bit is cleread when an operation is triggered. Note that no operation must be triggered when this bit is high. Otherwise, the result is unpredictable. However, a hardware failure interrupt should normally be generated. |

# 6 Creating and using soft devices

To create a soft device, we first need to map an area of memory which we are going to share with UML. This can be done with the `mmap` system call as follows:

```
device_resources.base_address =
        (unsigned char *)mmap (NULL, PAGESIZE, PROT_READ | PROT_WRITE,
                    MAP_SHARED,
                    device_resources.fd, 0);
```

Where `device_resources.base_address` is the base address of the shared mapping and `device_resources.fd` is the file descriptor specifying the file to be passed to UML, so that it can be mapped inside UML. Once we have this memory area, we can assign a layout for it, depending on the specification of the device we are implementing. For example, for the coffee machine, we lay out the registers starting from offset 0:

```
reg_cr  = (union  CR *) device_resources.base_address;
reg_isr = (union ISR *) ((unsigned int) (device_resources.base_address) + 1);
reg_dwr = (union DWR *) ((unsigned int) (device_resources.base_address) + 1);
reg_hsr = (union HSR *) ((unsigned int) (device_resources.base_address) + 2);
reg_csr = (union CSR *) ((unsigned int) (device_resources.base_address) + 3);
reg_acc = (__uint8_t *) ((unsigned int) (device_resources.base_address) + 4);
reg_idr = (union IDR *) ((unsigned int) (device_resources.base_address) + 5);
```

We then install a handler for `SIGUSR1`, so that when the state of the registers changes, we can perform the necessary action.

To be able to generate interrupts, we need to create an object which can be used to dispatch `SIGIO` signals to UML. In the coffee machine, and in Saint, we use unix domain sockets. We create a socket as follows:

```
sock = socket(PF_UNIX, SOCK_DGRAM, 0);
addr.sun_family = AF_UNIX;
snprintf(addr.sun_path, 5, "%5d", getpid());
err = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
```

We generate interrupts by writing to or reading from the socket:

```
    char c='1';
    err = sendto(fd, &c, 1, 0, (struct sockaddr *) &sun, sizeof(sun));
```

I/O memory can be implemented in soft devices in much the same way as I/O ports are, with the difference that there are no side effects associated with reading or writing to I/O memory, and the device must poll for changes.

# 7 Accessing soft devices from User-mode Linux

Device drivers for soft devices can now be written in UML just as they would be on the host system. Registers can be accessed using I/O instructions, for example:

```
void push_milk()
{
    reg_idr.bits.IDO=1;
    reg_cr.bits.PS=3;
    outb(reg_cr.byte,ADD_CR);
    down_interruptible(&op_sem);
}
```

Device interrupts can be intercepted by registering an interrupt handler, as shown below:

```
result = request_irq(CAFE_IRQ, cafe_interrupt_handler,
            SA_INTERRUPT, "cafe", (void *) last_status);
```

User processes in UML can use the `ioperm` and `iopl` system calls to gain access to a range of ports, and invoke the I/O instructions, `in` and `out` to read and write them as usual.

# 8 The Saint language

Before starting on a new language, we had the option of using Devil [6, 7], a language used to write device drivers. However, we decided that we required only a subset of Devil, and had some additional requirements for which we would require extensions. Thus, we designed the Saint language. The purpose of the Saint language is to be able to generate code for common tasks in the implementation of soft devices. The Saint stub generator also produces code to perform consistency checks at run time depending on specified properties. For example, if a port is defined as *read only*, the generated code will generate an error condition if it is written to.

Figure 8 shows an example device specification in Saint. The `ports` section defines 80 ports, addressed from 0 to 100, and specifies their properties with respect to whether they can be read or written to. The `read`, `write` and `none` properties can be specified for one or more ports, or one or more bits for a particular port. This scheme is followed in the definition of I/O memory, registers and event handlers as well. For example, the second line in this section defines the port at offset 5 as readable and writable. The `mem` section follows the same

```
device example {
    ports (0..80) {
        (0..4)  read;
        (5)     read,write;
        (6..79)  read;
        (80)   {
            (0..3) read;
            (4..7) write;
        }
    }

    mem (0..100) read,write;

    registers {
        ports {
            (0) ACC;
            (1) ISR;
            (2) {
               (0..1) STATUS;
               (2..7) ERR;
            }
        }
    }

    handlers {
        ports {
            (0..4) handle_first_four_ports(ACC,ISR,STATUS,ERR);
            (80) {
                (2..7) handle_error(ERR);
            }
        }

        faults handle_fault;
    }
}
```

semantics as the `ports` section, and is used to define I/O memory. In this case, it defines 100 memory locations addressed from 0 to 100, and that they can be read or written to.

The `registers` section defines named registers which will be passed to event handlers defined in the *handlers* section. Registers can span a range of ports, or a range of bits in one port. For example, the fifth line defines the register ERR to span 6 bits (2-7) of port 2. The `handlers` section defines a set of handlers for when ports are read and written to. The first line of this section specifies that the *handle_first_four_ports* function be called with the registers ACC, ISR, STATUS and ERR as arguments. The last line of this section specifies that the `handle_fault` function should be called in case of faults, such as writes to read-only ports or memory.

Figure 8 gives an example of generated stub code. The `handle_first_four_ports` function will be called when one of the first four ports is written to, after some consistency checks. Registers are maintained internally as structure following the C bit notation, where properties are defined for bits. For example, for port 2, we have:

```
struct __p2 {
    unsigned int STATUS : 2;
    unsigned int ERR    : 6;
};
```

```
int handle_first_four_ports(unsigned int &ACC, unsigned int &ISR,
                                    unsigned int &STATUS, unsigned int &ERR) {
      /* fill in your code here */
}
```

## 9 Related work

Although the testing of device drivers is an important concern in most Operating Systems, generic tools and methodologies to do this are surprisingly few. Microsoft provides one such set of tools [2] to perform some performance and compatibility tests on device drivers for Windows.

Devil [6, 7] is a language for generating device driver stubs in a safe and efficient way. Blue Water Systems' WinDK [5] and Compuware's NuMega [1] are other tools to generate device driver stubs. Saint on the other hand, generates stubs for soft devices, which encapsulate the semantics of the device in a limited manner.

## 10 Conclusion and Future work

We have implemented a limited framework for soft devices. We intend to build on this framework, partly by extending the Saint language to be able to define more complex interfaces, such as PCI, ISA, USB *etc.*. In the context of Saint, we would also like to be able to add more functionality so as to be able to describe entire soft devices instead of only interfaces to generate stubs. We would also like to implement some real soft devices to better understand some of the topics we have broached, such as profiling hardware devices. Finally, we would like to explore possibilities of a systematic tool set to be able to evaluate the efficiency of device drivers, and compare the performance of different device drivers using soft devices.

## 11 Acknowledgments

We would like to thank Jeff Dike, the creator of UML, for helping us resolve some UML related issues during the course of this project and Brian Code and Hedi Hamdi for their useful comments.

## References

[1] Compuware Corporation. Driverwork's user guide. www.numega.com.

[2] Microsoft Corporation. Windows device driver development kits. http://www.microsoft.com/whdc/ddk/winddk.mspx.

[3] Jeff Dike. User-mode linux. In *Proceedings of the Ottawa Linux Symposium*, 2001.

[4] Jeff Dike. Making linux safe for virtual machines. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[5] Blue Water Systems Inc. Windk users manual. www.bluewatersystems.com.

[6] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, October 2000.

[7] L. Réveillère and G. Muller. Improving driver robustness: an evaluation of the Devil approach. In *The International Conference on Dependable Systems and Networks*, pages 131–140, Göteborg, Sweden, July 2001. IEEE Computer Society.