Implementing a User-Mode Linux with Minimal Changes from Original Kernel

Hans-Jörg Höxer Kerstin Buchacker Volkmar Sieh

Institut für Informatik 3
Friedrich-Alexander-Universität Erlangen-Nürnberg
Germany

info@umlinux.de

Abstract

This paper presents some aspects of implementing a User-Mode Linux with as few changes to the original Linux kernel as possible. To port a Linux kernel to a User-Mode environment, basically all parts of the kernel directly interacting with the hardware must be changed. To accomplish this, we need an environment which simulates some hardware parts. This includes simulation of interfaces to device controllers such as keyboard, IDE, graphics or network controller and other devices such as the real-time-clock. Secondly a solution must be found for replacing the assembler code contained within the original kernel with functions. The implementation of the functions, which are called instead of the assembler code, should not become part of the kernel, but be part of the User-Mode environment, just like the implementation of the assembler instructions is not part of the kernel but part of the CPU.

1 Introduction

Our team is implementing a User-Mode Linux called UMLinux with three main targets in mind. The first one is, that the changes needed to port an original Linux kernel to our User-Mode environment should be minimal. The second is, that it should be possible to make the virtual hardware the User-Mode Linux is running on fail at will. We plan to use this capability to test how well fault-tolerant systems such as the Linux Virtual Server described at `www.linuxvirtualserver.org`, can handle failures of hardware components. And thirdly, an automatic experiment controller should make it possible to run lengthy experiments without user interaction.

The first goal, minimal changes from the original kernel, is the main topic of this paper. The advantages are clear. The fewer lines of code we touch in the original kernel, the fewer errors we can introduce into the kernel and the less effort is needed to port the kernel to UMLinux. This speeds up adapting UMLinux to new kernel releases.

The parts of kernel code that need to be changed, are those directly interacting with the hardware. Basically these are all lines containing inline assembler instructions. Assembler is used in three main areas of the kernel: for interrupt and exception handling, to access functions of the memory management unit and for communication with devices via `in*/out*` instructions. Our approach is to replace these lines of assembler code with functions, which have the same effect as the original assembler instructions. Since the implementation of the assembler instructions called in the original kernel are not part of the kernel itself but of the CPU, the functions replacing these assembler instructions should not be part of the kernel either, if changes and additions to the kernel are to be kept minimal. These functions are therefore provided by the User-Mode environment.

In UMLinux, signals replace interrupts and exceptions. Consequently, signal handler functions replace interrupt and exception handlers.

The signal handler functions in UMLinux do not actually contain code to handle the interrupt. The signal handler function modifies the stack so the original kernel's interrupt handler can work with it and then calls that handler. Upon return of the handler, the modifications are undone by a function implementing the `iret` assembler instruction and execution of the User-Mode kernel continues.

Assembler instructions concerning the memory management unit are those, for example, which modify the page-directory base register or segment registers. We have implemented functions based on the system calls `mmap/munmap` to replace these instructions and simulate the memory management unit.

Communication with a number of devices is done via interrupts and `in*/out*` assembler instructions. UMLinux therefore provides functions to replace the `in*/out*` instructions. These functions implement the UMLinux virtual hardware devices, such as the real time clock, IDE-controller, keyboard-controller and network-controller. These virtual devices communicate with the UMLinux kernel just like real devices would communicate with the real kernel. Consequently we can use the device drivers of the original kernel in UMLinux, too. Changes necessary to port the Linux kernel to UMLinux are therefore reduced to changes in the architecture dependent files in `include/asm` and `arch` only. No new drivers are needed.

It is not our goal, to create a complete hardware simulation like Plex86 [6]. We just want to simulate enough of the hardware, to be able to use the original Linux device drivers. Enough of the hardware, in our case, means providing the hardware interfaces the Linux kernel uses and the functionality provided by the hardware.

The common problem all User-Mode kernel implementations have, is how to make compiled program binaries trap into the User-Mode kernel when making a system call. Since the binaries are identical to those running on the real kernel, each time when making a system call, they would trap directly into the real kernel instead of the User-Mode kernel. To avoid this, User-Mode implementations must devise a mechanism to catch all system calls made by binaries running on the User-Mode kernel and divert them to the User-Mode kernel. How this can be accomplished is described in some of our papers [1, 7] and some of Jeff Dike's papers [4, 5]. The basic idea is to trace the process executing the User-Mode kernel, stopping it whenever it makes a system call and having the tracing process divert the system call using the functionality provided by the `ptrace` system call. The papers cited above also describe in more detail as is done here how the basic hardware, such as random access memory (RAM), storage devices and network interfaces can be implemented. To understand the following discussion, it should suffice to know, that a UMLinux machine's RAM is implemented as a memory mapped file, storage devices are files and network interfaces are sockets.

The rest of the paper is structured as follows: Section 2 gives an overview of the User-Mode environment in which the User-Mode kernel runs. Section 3 explains in a little more detail how the User-Mode kernel interfaces with the virtual device controllers. Next, exception handling is treated in section 4. Memory management functions are the topic of section 5. Section 6 explains what can be done about privileged assembler instructions which occur outside the kernel and were therefore not replaced by calls to UMLinux simulator functions. Section 7 concludes the paper.

2 Accessing Simulation Code from the Kernel

The functions replacing the assembler instructions used in the original kernel are made available to the UMLinux kernel in a very roundabout way. This section explains how this is done and why we chose this approach. Section 2.1 gives an overview of the binaries needed in the UMLinux environment and the role of each of them. Section 2.2 treats the processes involved and section 2.3 shows how it is all put together to make the necessary code available to the UMLinux kernel.

2.1 The Binaries

Three binaries are involved. One is of course `vmlinux`, the binary containing the UMLinux kernel code. The next is `simulator`, which contains all the functions replacing the assembler instructions in the original kernel. Finally there is `node`, which includes the code for the graphical user frontend (GUI), some initialization code and the tracer.

The most visible part of the GUI is the UMLinux machine's console, which simulates this machine's monitor and keyboard. Whenever the (real) cursor is inside the (virtual) console, input from the (real) keyboard and mouse is directed to the UMLinux machine. Apart from that the most important buttons the GUI provides are those to power the UMLinux machine on and off.

The initialization code prepares the virtual hardware of the UMLinux machine. This includes creating the file used as RAM, opening the sockets for network interfaces, and creating hardware such as graphics controller and advanced programmable interrupt controller (APIC) if available. Both the graphics controller and APIC have their own memory, which is also implemented as a memory mapped file. These memory-files are created and possibly filled with configuration information at this stage. UMLinux machines have a minimal BIOS. The only BIOS functions provided are a function returning the size of the physical memory of the UMLinux machine and a function returning the start address of the video buffer. Most of the rest of the memory filled with BIOS functions in a real machine is taken up with the UMLinux simulator code. The hardware is present even before the machine is turned on, and some data may be stored statically on board (e.g. hardware setup information or read-only memory of extension cards). When the UMLinux machine is powered on, the hardware processes are signaled to start executing their respective tasks.

The tracer has the task of tracing the UMLinux machine process and catching and diverting the system calls processes on this machine make to the UMLinux kernel.

2.2 The Processes

Obviously at least two processes are needed: one to run the UMLinux machine and one to run the tracer. Another process is needed to run the GUI. This is actually the first process started. As is shown in figure 1, this process forks to create the processes simulating the virtual hardware (solid or dashed arrows in figure1). If present, the graphics controller and APIC will be created as separate processes. The tracer creates the processes for the virtual processors, as it needs to trace these processes. It also creates the APIC-process, because the APIC needs information about the number and ID of processors. There is no need to have a separate process for each (IO-)APIC, a single process — created as soon as one APIC is present — simulates all (IO-)APICs. Until the user powers on the UMLinux machine, these processes just sit there doing nothing (just like the real hardware before being powered on). When the user powers on the UMLinux machine, the original process signals (dotted or dash-dotted arrows in figure 1) the hardware processes to start doing whatever they should be doing. The processors will now finally load the UMLinux kernel. If the kernel parameters specify an `initrd`, this will be loaded, also.

Processes shown as boxes with solid lines are necessary for a minimally configured UMLinux machine with a single processor. Those shown as boxes with dashed lines are optional and their creation depends on the configuration of the UMLinux machine. The names given to the processes in figure 1 are descriptive of the role, they are not the name of the binary being executed. The name of the binary being executed is shown to the right of the descriptive name. Of course the CPU# processes also execute any number of user processes running on the UMLinux machine in addition to `vmlinux` and `simulator`. The binaries for the UMLinux user processes are loaded from the UMLinux harddisk.

For some of the processes their exact position in memory is important to make UMLinux work. The following paragraphs therefore digress shortly to explain the memory layout of physical and virtual memory.

The layout of the physical memory must be distinguished from the layout of the virtual memory. Figure 2 shows the layout of a real and a UMLinux machine's physical memory (top and bottom). The address range of physical memory is from zero to several GB in Intel x86 processors. Of course, in most cases, only a few hundred MB are indeed installed in the system, i.e. there is a gap in physical memory. The memory shown towards the high end of addresses (named SVGA in figure 2) is usually physically part of the on board memory of the graphics card. The box at the bottom of figure 2 shows the layout of a UMLinux machine's physical memory, which is (except for perhaps the size of the available physcal memory) basically identical to that of the real machine's.

The center box of figure 2 shows the layout of the real machine's virtual memory after a UMLinux machine has been started. The user processes (both those of the UMLinux machine

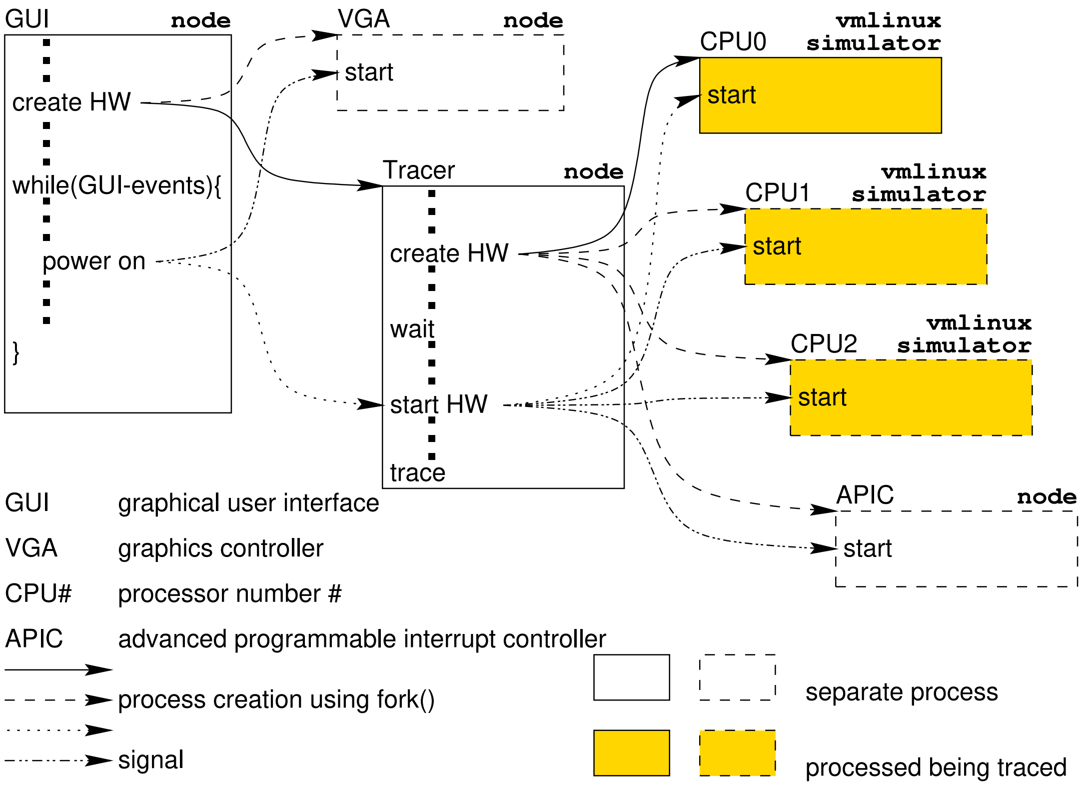

Figure 1: Processes

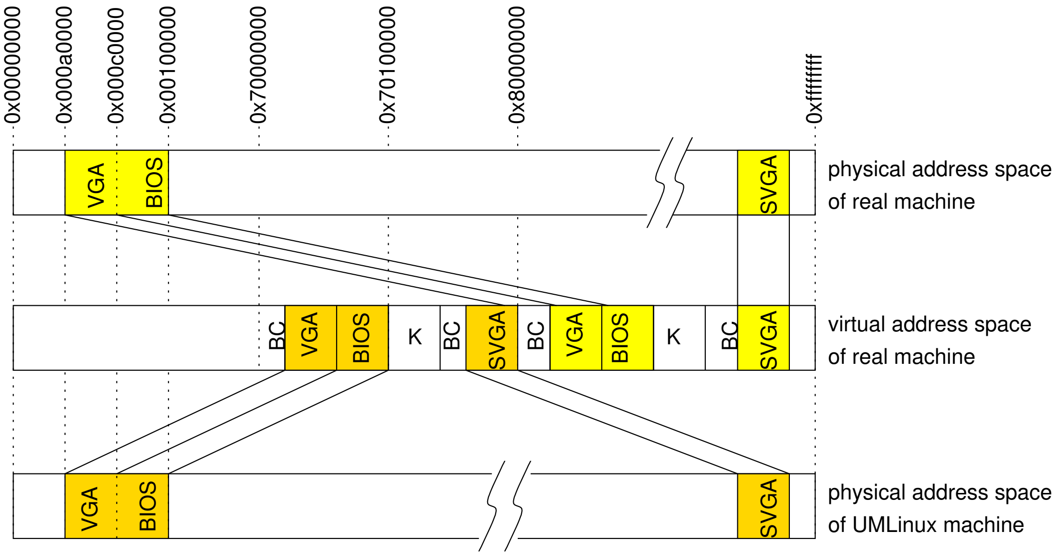

Figure 2: Memory Mapping

and those of the real machine) are mapped into the lower part of memory, the higher part (depending on the Linux kernel version and compilation parameters usually beginning at address 0x8000000 or 0xc0000000) being reserved for the kernel (K). As is shown in figure 2, some hardware-dependent parts of the physical memory, such as BIOS and VGA, are mapped into the kernel space. The kernel uses the remaining memory in kernel space for buffer caches (BC). The UMLinux kernel space is mapped beginning at 0x70000000. Its internal layout is similar to that of the real kernel. The UMLinux machine loads the UMLinux kernel (`vmlinux`) to the appropriate address and maps space for BIOS and VGA as needed.

All processes shown in figure 1 are real user processes. As such, they start executing in the real machine's virtual memory space reserved for user processes. The CPU#-processes will load the UMLinux kernel and therefore execute code in UMLinux kernel space part of the time. Since these processes simulate a CPU, they will of course also execute code in user space, too. In addition, they must be able to access the simulator code whenever necessary.

2.3 Making Simulation Code Callable from the Kernel

The UMLinux kernel needs to access the simulation code, whenever it wants to execute one of the set of simulated assembler instructions. This code must therefore be made available at a fixed address in memory. For UMLinux, we have decided to put the simulation code with the BIOS, which is mapped in kernel space at a fixed address. We did not want to include the simulation code directly with the kernel, as we believe it is very important to change the original kernel as little as possible. Nevertheless, the kernel must be told where to find the entry points to the functions replacing the needed assembler instructions.

As described in section 2.1, the simulation code is in a separate binary, `simulator`. This binary is converted into a datastructure (which is simply a character array containing the hexdata of the binary bytewise). This datastructure is then in fact compiled into the `node` binary. During the hardware creation of a UMLinux machine, this data is written into the memory mapped file which simulates the BIOS PROM. The file is mapped to the correct address during the power on process of the UMLinux machine.

3 Communicating with Devices

There are basically two possibilities the processor has to communicate with devices, memory mapped I/O and the special I/O space. Addresses in the I/O space can only be accessed with `in*/out*` instructions, whereas addresses in memory mapped I/O can simply be accessed via `mov*` instructions. In both cases, registers on the device mapped to that address range are accessed. Devices mapping their I/O ports to the I/O space include the keyboard, IDE, VGA and NE2000 network controller as well the real time clock. The VGA controller uses I/O space only for control registers. The video buffer is memory mapped. The APIC and I/O APIC also use memory mapped I/O.

The following sections explain how communication with devices is implemented in the two cases.

3.1 Memory Mapped I/O

The problem with memory mapped I/O is the fact, that it is accessed with `mov`, one of the most commonly used of all assembler instructions. Unlike the `in*/out*` instructions, which are mostly used by the kernel and seldomly by application programs, `mov` is also used by basically all applications programs. Since we do not touch application code to port it to UMLinux we cannot replace `mov` instructions in application programs. Catching `mov` instructions in some way would definitely lead to a massive performance loss.

The most important device using memory mapped I/O is the VGA controller. We decided, to implement the VGA controller as a separate process (see also figure 1), which continually scans its memory mapped I/O and displays any changes on the UMLinux machine's monitor. Using this method is fine when no immediate reaction to a change in some byte in the memory mapped I/O address range is needed and no reply is expected from the device.

For other devices using memory mapped I/O, for example the APIC, we have replaced the macros and inline assembler instructions in the original code with wrappers calling our replacement functions instead.

3.2 Using the I/O Space

The situation is a little different for devices communicating through the special I/O space. Applications do not normally use the `in*/out*` instructions. Most occurrences of these instructions are in the kernel, where they can easily be replaced. To keep the assembler code centrally in one place, there is a file `asm/io.h` in the original kernel sources, which defines a number of macros in order to access the devices' I/O ports via the `inb/inw/inl` and `outb/outw/outl` assembler instructions and their string versions `insb/insw/insl` and `outsb/outsw/outsl`.

To port to UMLinux using our simulation facilities, we simply change those macros, which directly use the assembler instructions, to use the UMLinux simulation functions instead. That's all that needs to be changed in the original kernel sources.

To make it all work, the UMLinux simulator must provide all of the assembler instructions listed above. We have implemented one function for each of the assembler instructions (named after the instruction). These functions are passed the I/O port as a parameter (just like the assembler instructions [2]). As is shown in figure 3, the functions branch internally according to the I/O port number to call the function appropriate to the device accessed via the given I/O port. The functions implementing the `in*` instructions return a value, the functions implementing the `out*` instructions do not. For a number of devices (e.g. PIC, IDE, network controller) the specific function is called with additional parameters depending on the port used to access the device. There are several branches for these devices (not shown in figure 3).

The `in*/out*` functions are the entry points into the simulator. In a real machine, the device controllers usually execute asynchronously to the program being run on the processor. After reading/writing data from/to a controller's I/O ports the processor continues execution with the statement following the `in*/out*` instruction. Meanwhile the device controller handles the request. When the data is ready, the controller makes it available in its I/O ports and an interrupt is generated to let the processor know, that data has arrived. Figure 4 summarizes the timing. In some cases, the request is handled synchronously. An example is reading the time counter from the real time clock. To do this, the processor writes a latch request to the real time clock's I/O ports and then reads the returned data with an `in*` instruction.

In a UMLinux machine things are quite similar. Executing `out*()` from somewhere in the kernel calls the `um*_out*()` for the appropriate device and control passes to the device controller simulation code to handle the request. If necessary, the function `isa_irq()` is called to generate an interrupt. Control is then passed back to the interrupt handling procedure in the kernel sources in a very roundabout way. Please refer to section 4 for details on interrupt handling. As is shown in figure 5, control is passed back and forth between the `vmlinux` and the `simulator` binary during this phase. The notable difference between the handling of I/O operations on a real machine and on UMLinux, is the fact, that in UMLinux I/O operations are handled synchronously. Figure 5 shows, that the kernel is stopped while the device controller code is executed. The kernel running on a UMLinux machine is therefore interrupted right at the instruction following the `out*` instruction, where the kernel running on a real machine will usually have executed some more code before code before it receives the interrupt generated by the I/O request.

Some applications, like the X-server, do use `in*/out*` instructions. Since we only port the Linux kernel to UMLinux and do not touch application programs and binaries, we cannot stop the X-server from using `in*/out*` instructions. How we handle these cases is explained in section 6.

4 Interrupt and Exception Handling

We can distinguish two mechanisms for interrupting normal program execution, interrupts and exceptions. Interrupts are asynchronous events, usually triggered by an I/O device. Exceptions, on the other hand, are synchronous events which are generated when the processor detects a special predefined condition (such as a division by zero) while executing an instruction. Interrupts and exceptions are also generically called traps. In both cases, normal program execution is interrupted and execution of a han-

```
unsigned char inb(unsigned short port) {
    if (0x0020 <= port && port <= 0x0021) {
        value = umpic_inb(0, port - 0x0020);
    } else if (0x0022 <= port && port <= 0x23) {
        value = umapic_inb(port - 0x0022);
    } else if (0x0060 == port || port == 0x0064) {
        value = umkbd_inb(port);
    } else if (0x0070 <= port && port <= 0x007f) {
        value = umrtc_inb(port);
    } else if (0x0160 <= port && port < 0x0168) {
        value = umide_inb(5, port - 0x0160);
    } else if (0x0280 <= port && port < 0x02a0) {
        value = umne2000_inb(1, port - 0x0280);
    } else if (0x03c0 <= port && port < 0x03db) {
        value = umvga_inb(port);
    } else if ...
        ...
    } else {
        printf("reading byte from unassigned port 0x%04x\n", port);
        value = (unsigned char) in_res++;
    }
    return value;
}
```

Figure 3: Simulation of the in*/out* Assembler Instructions

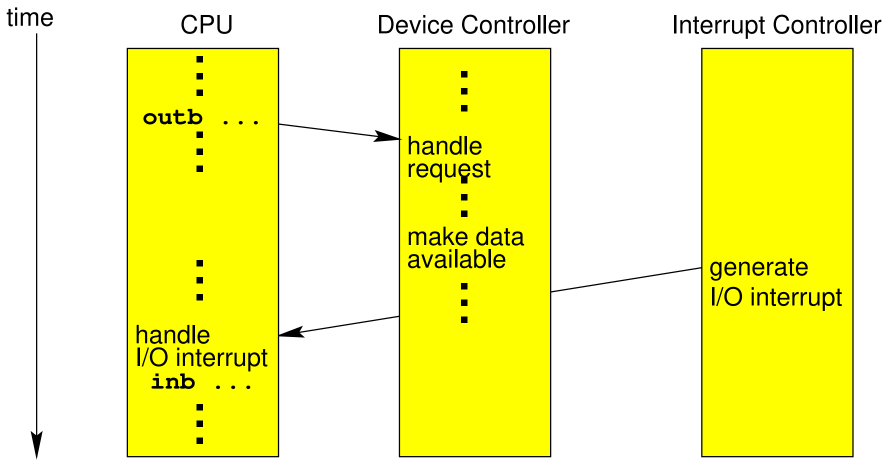

Figure 4: Timing of I/O Operations on a real machine

dler procedure is started. When the handler procedure has finished (possibly after being itself interrupted), normal program execution continues at the instruction following the one where the program was interrupted. The entry points into the trap handling procedures (address of the first instruction) are stored in the interrupt descriptor table.

Trap handling is supported by hardware mechanisms. When a trap needs to be handled, the processor does the following [2]:

1. push the current contents of the registers containing flag information, code segment selector and instruction pointer onto the stack.

2. if appropriate, push an error code onto the stack.

3. load the segment selector for the code segment containing the trap handling code and the new instruction pointer into the appropriate registers.

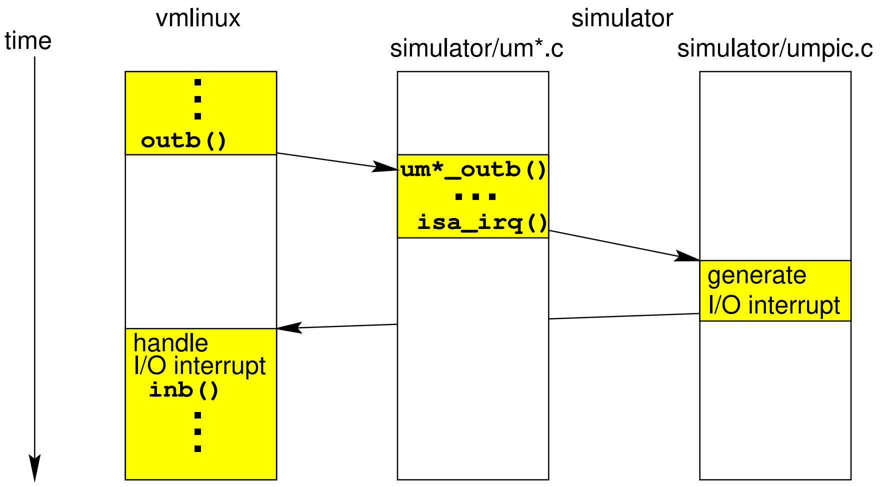

Figure 5: Timing of I/O Operations on a UMLinux machine

4. if the trap was an interrupt, disable interrupts during execution of the handler.

5. begin execution of the handler procedure.

It is possible to use a different stack for the execution of trap handling procedures. If this is the case, a stack switch occurs when switching from normal program execution to a trap handling routine. The processor then has to temporarily save the registers containing the stack segment selector and the stack pointer in addition to those listed in 1. Next, the processor loads the stack segment selector and stack pointer of the new stack into the appropriate registers. The new stack is used in all subsequent operations involving the stack and the machine proceeds with steps 1 to 5 (the saved register contents are pushed in step 1).

To return from a trap handler the assembler instruction `iret` is used. This instruction is similar to the `ret` instruction used to return from a normal function call, except that it also restores the flags register for the interrupted function. To return from a trap handler, the following steps are necessary:

1. restore the registers containing the code segment selector, flag register and instruction pointer to the saved values.

2. pop values saved when entering into trap handler from stack

3. resume execution of interrupted program.

If the stack was switched when calling the trap handler, a switch back to the previous stack is accomplished in step 2 by restoring the previous values of the stack segment selector and stack pointer instead of simply popping the stack.

UMLinux interrupts can be of two kinds. One is a signal. Whenever an interrupt arrives for a user space process, Linux sends that process the appropriate signal, e.g. when the UMLinux machine requests a page not in memory, the hardware (of the real machine) will generate a page fault exception and the Linux operating system (of the real machine) will send a `SIGSEGV` to the user process that caused the page fault exception. When the process executes an `int3` instruction, it is sent a `SIGTRAP`. The other is an interrupt generated by the simulator itself using `isa_irq()` (as was described in section 3.2). This function simply executes an `int3` instruction (which will raise a `SIGTRAP`).

Since a UMLinux machine runs as normal user processes, we do not have access to the hardware mechanisms of the real machine which support trap handling. Therefore, we have implemented these mechanisms in our simulator code. To ensure that our simulator mechanism is called whenever a trap occurs or a trap handler returns, we have to be notified whenever such a situation arises.

Remember that the CPU# processes, which execute the code in the `simulator` binary, are traced (figure 1). This means, the CPU# processes are stopped every time they execute a trap instruction (such as `int3`, `int n` [2]), receive a signal or enter/return from a system call, and the tracer then gains control.

To implement interrupt handling support in

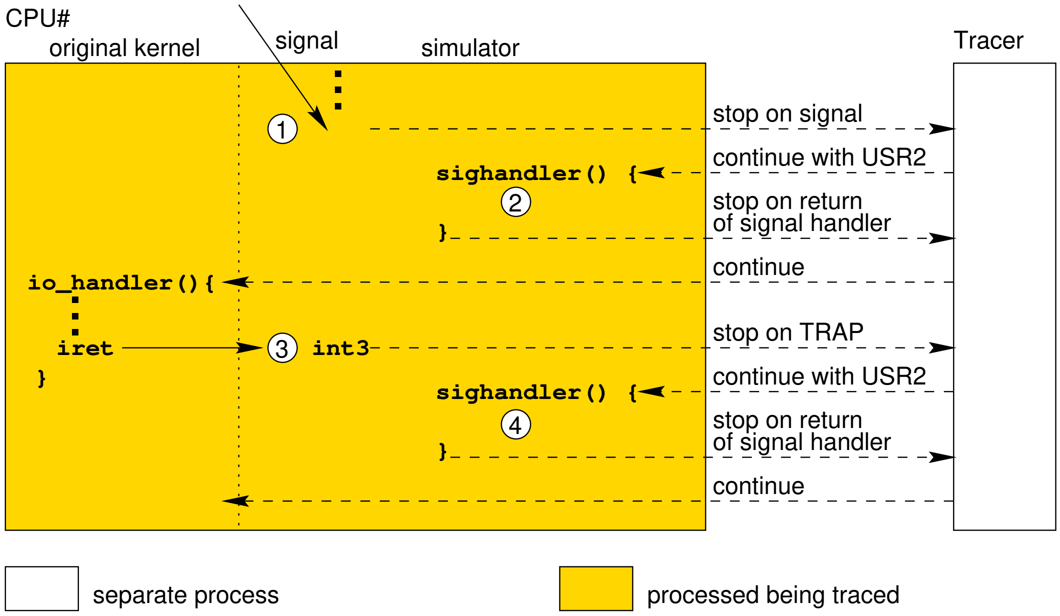

Figure 6: Interrupt Handling

UMLinux hardware, we just have to be able to stop the CPU# process in the right places and have the tracer manipulate the CPU# process to execute the simulator code that prepares the stack for the handler procedure and rebuilds it for resuming the normal program code. A somewhat simplified flow of control during interrupt handling on a UMLinux machine is shown in figure 6.

When an interrupt arrives, it can either be one generated by the function `isa_irq` from section 3 or a signal (number 1 in figure 6). To stop on return of a trap handler, we have changed `arch/i386/kernel/entry.S` to call `int3` instead of `iret` (number 3 in figure 6. (`arch/i386/kernel/entry.S` contains system-call and low level fault handling code.) When the tracer gains control it checks whether the CPU# process is really stopped because of an interrupt/return of an interrupt handler. If this is the case, it lets the CPU# continue with a `SIGUSR2`. The `SIGUSR2` signal handler is part of the simulator code and implements the mechanisms to jump into/out of an interrupt handler. The `SIGUSR2` signal handler is the core of UMLinux trap handling and contains code to handle entering and leaving interrupt handlers. At number 2 in figure 6 the stack is prepared for entering a trap handler, at number 4 the stack is prepared for resuming normal program execution after returning from a trap handler. The CPU# process stops on return from a normal signal handling routine like the `SIGUSR2` signal handler from our simulator, because a `sigreturn` system call is automatically executed when returning from a signal handler (to clean up the stack frame). The execution of a system call causes a traced process to stop.

Of course figure 6 is somewhat simplified and does not show, for example, what happens, when other interrupts arrive while the kernel is executing a trap handler or how the changing of privilege levels is implemented.

5 Memory Management

The Intel system architecture supports virtual memory through paging. This means, that the linear virtual address space is divided into fixed size pages that can be mapped to physical memory or disk storage. When a process accesses a virtual memory location, this is translated into a physical address using the paging mechanism. If the page containing the virtual memory location is not currently in physical memory, a page fault exception is generated. The exception handler must load the missing page. On return of the handler, the operation which caused the page fault exception is restarted. The information necessary to map virtual memory locations to addresses in physical memory and (if necessary) to generate page fault exceptions is located in page directories and page tables stored in physical memory.

The base physical address of the page directory is contained in control register 3 [3] (page directory base register). Only the operating system is allowed to load new values into the control registers, using the `mov cr` instructions. We have therefore replaced all reads and writes of the page directory base register with simulator functions. Since the UMLinux machine runs as a normal user process without special privileges, it cannot write the real machine's page directory base register. The current value of the UMLinux machine's page directory base register is therefore stored in a data structure kept by the simulator. It is this value that the kernel running on a UMLinux machine accesses.

The translation of most recent virtual memory locations accessed are cached in so called translation lookaside buffers. Whenever the page directory base register is loaded with a new value, these buffers are invalidated. In UMLinux machine, the simulator uses `munmap` to unmap all pages from memory. All subsequent accesses to this address range will generate invalid memory references (page faults). The CPU# process will therefore be stopped with a `SIGSEGV` basically on the next instruction. The tracer simply lets the CPU# process continue and the CPU# process' handler for `SIGSEGV` is executed. This handler checks which virtual address was accessed and uses `mmap` to map a page from the appropriate file (e.g. UMLinux machine's physical memory file or UMLinux video memory file) into memory. In this way, only those pages really needed are mapped and unnecessary overhead is avoided.

6 Treatment of Assembler Instructions In the Wild

In the previous chapters we have explained, which assembler instructions we have replaced with special UMLinux simulator functions. To make sure, that the kernel running on top of the UMLinux machine uses our simulator functions instead of the original assembler instructions, we have modified the original kernel source files containing such instructions.

What about code external to the kernel which uses such assembler instructions? We have mentioned the X-server in section 3, another probable candidate are kernel modules. Sources other than the original kernel sources are not modified and may therefore contain assembler instructions which should really be replaced. When the CPU# process tries to execute such an instruction, a general protection fault will be generated instead. The (real) Linux will send a `SIGSEGV` to the CPU# process, which will stop. The tracer lets it continue so it can execute the `SIGSEGV` handler.

This handler checks, whether the `SIGSEGV` is due to a page fault (see memory management section 5) or a general protection fault. This can be distinguished by the error code pushed onto the stack (see also section 4). If it is a general protection fault, the handler has a closer look at the instruction which caused this fault. If it is one of the instructions which should be replaced with the corresponding simulator function, this function is called. On return from the function, the (previously saved) instruction pointer is set to the instruction after the one that caused the general protection fault. The `SIGSEGV` handler then returns and normal program execution can resume. In case it is another instruction, the `SIGSEGV` is a real (UMLinux) hardware interrupt and the stack is prepared to call the appropriate interrupt handler of the original kernel (as described in section 4).

To run a completely unmodified kernel on UMLinux (such as is possible with VMware™[8]), the method described here would have to be refined. What we are currently doing by hand in the kernel sources would then be done automatically in the binary, before actually executing the code.

7 Conclusion

This technical paper explains which features a User-Mode simulator must provide to minimize modifications when porting the original Linux kernel to this User-Mode environment. Currently, our UMLinux simulator only runs on the Linux operating system. Once the kernel is solely dependent on the simulator interface, the User-Mode environment can be ported to other operating systems. We are planning to port it to the Microsoft Windows family of operating systems.

Acknowledgement

The research presented in this paper is supported by the European Community (DBench project, IST-2000-25425).

References

[1] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings Sixth IEEE International High-Assurance Systems Engineering Symposium*, pages 95–105, 2001.

[2] Intel Corporation. Intel architecture software developer's manual (volume 1: Basic architecture), 1999.

[3] Intel Corporation. Intel architecture software developer's manual (volume 3: System programming guide), 1999.

[4] J. Dike. A user-mode port of the Linux kernel. In *4th Annual Linux Showcase & Conference, Atlanta*, 2000.

[5] J. Dike. A user-mode port of the Linux kernel. In *5th Annual Linux Showcase & Conference, Oakland, California*, 2001.

[6] K. Lawton. Plex86. URL: http://www.plex86.org/, 2001.

[7] V. Sieh and K. Buchacker. Testing the fault-tolerance of networked systems. In U. Brinkschulte, K.-E. Grosspietsch, C. Hochberger, and E. W. Mayr, editors, *International Conference on Architecture of Computing Systems ARCS 2002, Workshop Proceedings*, pages 37–46, 2002.

[8] VMware Inc. VMware. URL: http://www.vmware.com/, 2001.