Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University

Virtual Operating System In C#

Bachelor's Thesis

Michal Burger

Informatics 9.2.1

Advisor:
RNDr. Richard Ostertág

Bratislava, 2008

Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University

Virtual Operating System In C#

Bachelor's Thesis

Michal Burger

Informatics 9.2.1

Advisor:
RNDr. Richard Ostertág

Bratislava, 2008

I hereby declare that I wrote this thesis by myself, only with the help of referenced literature, under the careful supervision of my thesis advisor.

....

Acknowledgments

I would like to thank my advisor RNDr. Richard Ostertág for the help and guidance he provided me with while I was writing my thesis. I would also like to thank him for suggesting the topic of this thesis and for everything he taught me during our Programming and Operating Systems classes.

Secondly I would like to thank Mgr. Vladimír Koutný for unknowingly sparkling in my mind the idea of writing a virtual operating system and for sharing with me all the materials he used while instructing us on the Nachos project in the Operating Systems class.

Abstract

Virtual operating system is a computer operating system which instead of on real hardware only runs on a simulated virtual computer. Virtual operating systems can be used for educational purposes but they also have their advantages compared to traditional operating systems, namely the improved security gained from separating the virtual processes from the real operating system and other processes running on the real hardware. Virtual computers also tend to have very simplistic design which sometimes allows us to prove the correctness of an operating system running on such computer.

In this paper I will introduce the Comenius Operating System (ComOS) and the virtual computer it runs on – ComOS Virtual Machine. The goal of the ComOS project is to create an environment in which students of Operating Systems classes can design their own operating systems and learn how to solve problems commonly encountered in system programming. Both ComOS and ComOS Virtual Machine are written in the C# programming language and post-compiled by PostSharp.

Keywords: ComOS, virtual operating system, PostSharp

Preface

Computers are devices designed to perform the same number of mathematical operations as would take an average human his lifetime to do, in just a fraction of a second and without single error. Early computers were used to break military codes, calculate rocket trajectories and simulate nuclear chain reactions. Even though most of the present day computers are used merely for word processing and browsing the Internet, it doesn't mean that computers have lost their importance. Quite on the contrary, the world as we know it could never exist without them.

But computers by themselves are just big metal and plastic boxes – they would be useless without software to run on them. And in turn all of the software would be useless too without the most important part of it – the operating system. Now a knowledgeable reader could object that the concept of an operating system didn't even exists at the time when first computers were built. But this reader should keep in mind that the early computers were very specialized devices that could often execute only a single program or had to be mechanically reprogrammed before every use. At that time there was no need for an operating system because the computers were very simple and only used for one and the same task. However, as the technology advanced, computers became mass produced and became one the most versatile and most complex devices man has ever created. Nowadays you can use the same piece of hardware for creating a short animated movie or for running a nuclear power plant. You could even do both at the same time! (Don't try this at home.) The original specialized programs were replaced by one general purpose program – the operating system, a program that allows you to run *other* programs rather than being useful by itself.

Since operating systems affect all other programs run on the computer, they are

designed with speed and performance in mind. For this reason the programming language of choice for a system programmer has always been the assembly language or language C. But as modern computers become increasingly more powerful, the performance of an operating system is not as important as it was in the past. The emphasis is now more on stability and security rather than efficiency and memory usage. And since the complexity of modern operating systems makes their development in low-level programming languages also very impractical, a simple solution comes to mind – using a high-level object oriented language. Engineers at Microsoft Research have already started exploring this idea in their managed operating system called Singularity, a research operating system written in C# (see [Sin]). This project takes advantage of provability of certain subset of the Microsoft Instruction Language to prove the correctness of parts of the operating system. Other community based projects exist that try to implement an operating system in the C# or Java languages, most notably SharpOS written in C# and JNode written in Java.
With all these new and exciting technologies in our hands, a high-level language operating system could soon become a reality. Most of the software development is already shifting from C and C++ to Java or the .NET framework and many young programmers learn Java as their first programming language. This is the reason why I decided to write ComOS in a managed language – not only to keep up with the latest technologies but also to contrast the modern high-level programming languages with the crude workings of actual computer hardware, to bring these two different concepts together and let the programmer see "how stuff works".

Michal Burger

Contents

Chapter 1

Introduction

In modern personal computers, operating system is the most important piece of software. It is the link between computer hardware and computer software. Operating system allows other programs to use computer resources and peripherals in a hardware-independent way, without worrying about the low-level implementation details. Most modern operating systems also allow execution of multiple programs at the same time while making this process completely transparent to the application programmer.

A virtual operating system accomplishes the same tasks as a regular operating system, only it doesn't run on a real hardware but rather on a virtual computer often called the virtual machine. Comenius Operating System (ComOS) is a project of writing an operating system for the ComOS Virtual Machine. In the following sections, I will present the rationale behind the ComOS project and provide a brief summary of the chapters to follow.

1.1 Motivation

The purpose of Comenius Operating System is to serve as an educational tool. It was specifically designed for the 1-INF-170 Operating Systems course at the Faculty of Mathematics, Physics and Informatics of Comenius University. There were only two requirements given for this project:

- It should allow students to implement the basic synchronization mechanisms, scheduling algorithms and virtual memory paging.
- Students should be able to write their code in a modern object-oriented language – specifically the language C#.

The reason for choosing the C# programming language in the second requirement was very pragmatical – to allow the shift from Java to C# in the 1-INF-225 Programming 3 course.

Since writing our own virtual operating system is a complicated task, the first thing we should do is to consider existing alternatives. As it turns out, there are many different systems used around the world for teaching Operating Systems courses that would fulfill out needs, had they not shared one common feature – the system code always has to be written in assembly or the C programming language. One particular project stands out from the rest in respect to this common property. Its name is NACHOS (Not Another Completely Heuristic Operating System) and it was originally written in C++ but later ported to Java as Nachos 5.0j at the University of California, Berkeley. (For more information about this project, see [HC01].) The Java version of Nachos has been successfully tested in the 1-INF-170 Operating Systems course and was the basis for Comenius Operating System.

However, it is important to note that the ComOS project doesn't copy Nachos in any way. The structure of Nachos virtual machine (if we may call it so) was merely an inspiration in the process of designing the ComOS Virtual Machine. Some of the flaws in Nachos were taken into account when designing ComOS and were fixed; we will compare some aspects of these two systems later on.

1.2 Structure of this document

In the following chapters, we will explore the workings of a real-life computers and compare them to the ComOS project. In Chapter 2, the concept of a virtual machine will be explained and the ComOS Virtual Machine will be compared and contrasted to the traditional virtual machines. Chapter 3 will deal with the architecture of modern computers and describe the simplified model used in ComOS Virtual Machine. It will also contain implementation details of some parts of the virtual machine. We will discuss the realization of ComOS software in Chapter 4 and study possible alternative uses for the ComOS project. Chapter 5 will conclude this paper.

Chapter 2

Virtualization

Virtualization is a broad term used for many different techniques in software development. In this chapter it will refer to the process of moving from a real computer hardware to a virtual machine.

2.1 The concept of a virtual machine

Virtual machine is a software implementation of a computer. It can run programs just like a real computer but it's not made of electronic circuits – it's just a software simulation, a program which in reality has to be run on another computer called the host machine. This host machine could in turn be again virtual – we shall shortly see that this in fact is the case of ComOS – but every such chain of virtual machines must ultimately end at a piece of real computer hardware where the actual code execution takes place.

The most important requirement put on a virtual machine is that the processes running inside of it must be limited to the abstraction provided by the virtual machine and won't be able to break out to the host operating system. This is often easily achieved by having the architecture, instruction set and binary format used in the virtual machine completely different to that of the host machine, thus making the virtual processes completely incompatible with the host operating system. This property of virtual machines is one of their greatest strengths – you can run potentially unsafe programs inside of a virtual machine without worrying about the stability or any other undesirable side effects to the host operating system.

Currently available virtual machines can be divided into two fundamentally different categories. The first one contains virtual machines which simulate existing real-world hardware with the purpose of running multiple instances of an operating system on one computer, possibly for security, compatibility or development reasons. A well known example of such virtual computer is the VMware Player. In the second category are the so called application virtual machines such as Java virtual machine and the Common Language Runtime. They don't emulate existing computer hardware but rather take advantage of the virtualization concept and define their own specification of a virtual computer. Their goal is to provide a highly abstract and portable runtime environment and they usually only allow to run one process per instance of the virtual machine.

2.2 Specifics of the ComOS Virtual Machine

The ComOS Virtual Machine falls somewhere between the two categories mentioned at the end of the previous section. It doesn't emulate any existing computer systems but provides it's own unique architecture, a very simplified model of what could a real computer hardware look like. It's purpose though is not to serve as a simple runtime environment. ComOS Virtual Machine needs to start an operating system before it can run any complicated applications. In fact writing any sort of application targeted at the ComOS Virtual Machine is not meant to be particularly easy. It's meant to be *hard*. The ComOS programmer will have to deal with many low-level aspects of computer hardware – probably not as many as would a real system programmer have to deal with but still enough to have to realize what is going on under the hood of an average personal computer.

There is one important difference between a real virtual computer and the ComOS Virtual Machine. A virtual computer should be capable of taking a program compiled into the virtual computer's machine code and execute it instruction by instruction. If we wanted the ComOS Virtual Machine to do the same thing, we would have to do one of the following:

- Specify our own instruction set, write a compiler to translate C# source code into machine code and also write an interpreter to read the machine code back and execute it.
- Use the Microsoft Intermediate Language (MSIL) as our machine language. We could then use existing C# compilers to compile code targeted at our virtual machine but we would still need to write an interpreter for the machine code (in this case for the MSIL).

As you can see, both options involve writing a machine code interpreter for a language with features such as automatic memory management, inheritance, polymorphism, function delegates and many others. This would certainly be a very difficult task to accomplish. For this reason I have decided to follow a different path. Since the ComOS Virtual Machine is itself written in the C# language, its host machine is the Common Language Runtime, a virtual machine capable of interpreting compiled C# code. All software written for the ComOS Virtual Machine is in fact a stand alone software written for the Common Language Runtime, it executes in the Common Language Runtime and only references the ComOS Virtual Machine as a library to utilize functionality provided by the machine. ComOS Virtual Machine only pretends that the code is executed inside the machine.

So what entitles us to say that this code really runs on our virtual machine? It's the restrictions that are put on the code. First of all the code cannot contain any system calls that would allow it to access the file system, console or other devices of the Common Language Runtime virtual machine. It also cannot create or manipulate threads and use synchronization primitives by any other means than the calls provided in the ComOS Virtual Machine. Secondly the virtual machine can directly control which ComOS threads are currently executing. It can also semi-preemptively switch the currently executing threads and it can keep track of how many instructions of certain kind have the individual threads executed. ComOS Virtual Machine in fact provides a very convenient way to control the code a ComOS programmer may write – for example, it allows him to use C# language features such as automatic memory management, events and exceptions but it won't allow him to print the result of

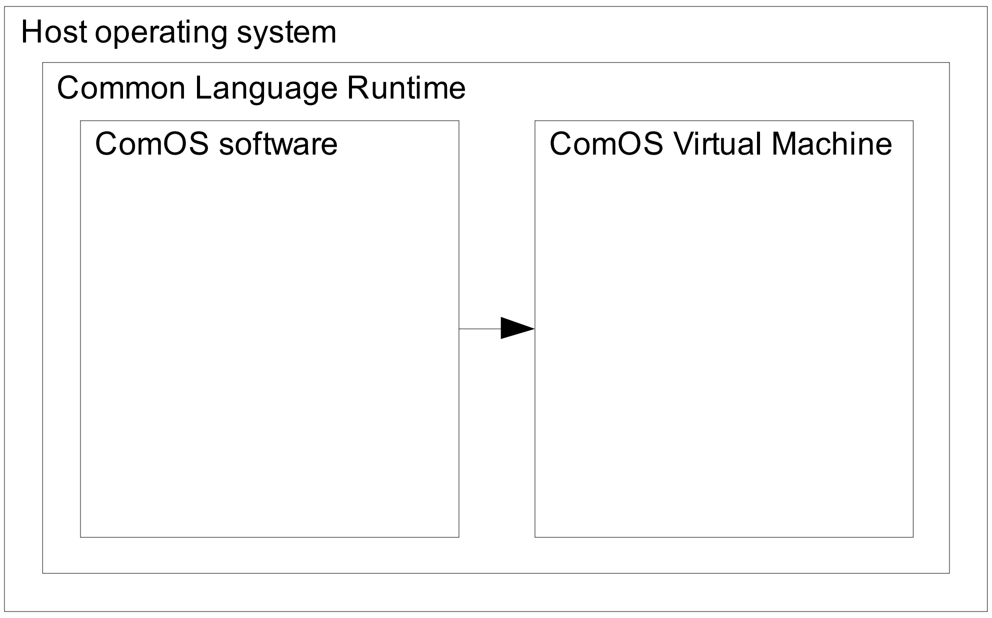

Figure 2.1: Relation between ComOS software and ComOS hardware

his computation to the screen by any other means than through the simulated ComOS console.

The question which comes to mind is how do we convince the programmer to not use any of the restricted calls that are otherwise an integral part of the C# programming language? And after playing with the code for a little while, even more complicated question arises: is it possible to preemptively control the execution of a managed thread at the MSIL instruction level? It wouldn't surprise me a bit if the answer was no. Fortunately there are other methods to work around this which we will talk about in Chapter 4.

Chapter 3

Designing a virtual computer

The purpose of the ComOS Virtual Machine is to simulate the functions of a real computer. For this reason we will first look at how modern computers work and then propose a simple computer model to be implemented in the ComOS Virtual Machine.

3.1 Architecture of modern computers

3.1.1 Von Neumann architecture

Design of most of the present day computers is based on the Von Neumann architecture of electronic computer as seen in Figure 3.1. Central part of this design is the processor

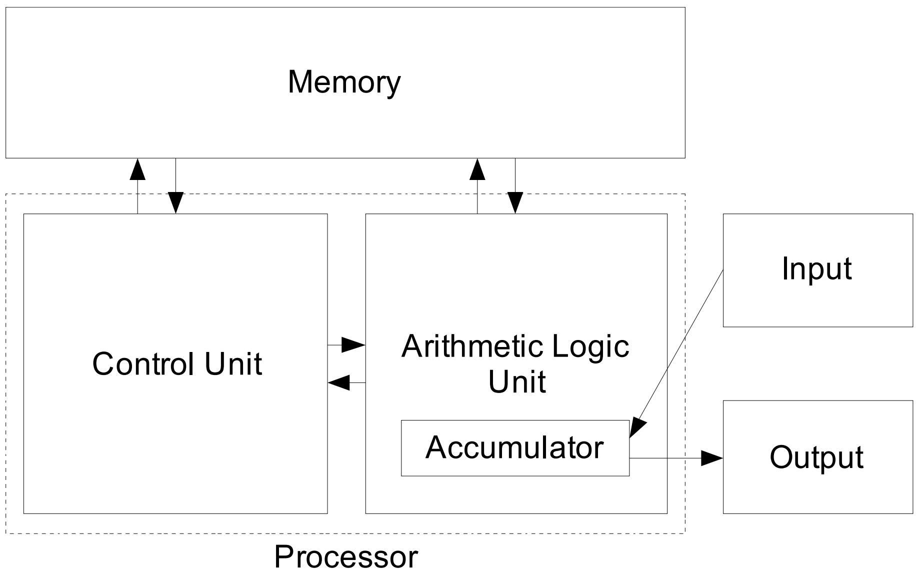

Figure 3.1: Von Neumann architecture

which consists of the Control Unit and the Arithmetic Logic Unit (ALU). Control Unit is a part of the processor which directs the flow of its operation. It fetches the instructions, decodes them, sends them for execution to other parts of the processor and then stores the results. Arithmetic Control Unit is a part of the processor which executes arithmetic operations. It uses the accumulator to store operands and results of these operations as well as data read from the input devices or data to be written to the output devices. Since the internal accumulator can only have a limited capacity, external memory is used for storing processed data. The memory also contains the program of the computer stored as a sequence of machine instructions.

3.1.2 Personal computer architecture

Although a typical personal computer (PC) may be based on the Von Neumann architecture, the actual implementation of this architecture can be many times more complicated. In Figure 3.2 we see the schematics of AMD-760™ MPX Chipset from the year 2000 [AMD01]. The features shared with Von Neumann model are the presence of processor (possibly more than one), memory and input / output devices. All these parts communicate through electric connections called buses, in the picture shown as black lines and arrows. Individual buses are then connected by bridges – the northbridge, in the diagram labeled as AMD-762™ System Controller, and the southbridge, labeled as AMD-768™ Peripheral Bus Controller.

If we carefully examine all the different buses and slots, we will notice that most of them serve the same basic purpose – to communicate data from processor to the peripheral devices and back. Why do we need such a big variety of device interfaces and buses? The reason is partly backward compatibility, partly different speed and throughput requirements. For example, we may notice that a graphics card in the AGP 4X Slot has a dedicated bus connecting directly to the northbridge because it needs to transfer huge amounts of data from the system memory. The system memory too is connected to the northbridge by a dedicated bus rather than sharing the same bus for example with a network card or a keyboard. Since the typical amount of data a device needs to process varies greatly with different types of devices, we need to have many different buses to accommodate the particular needs of all of them.

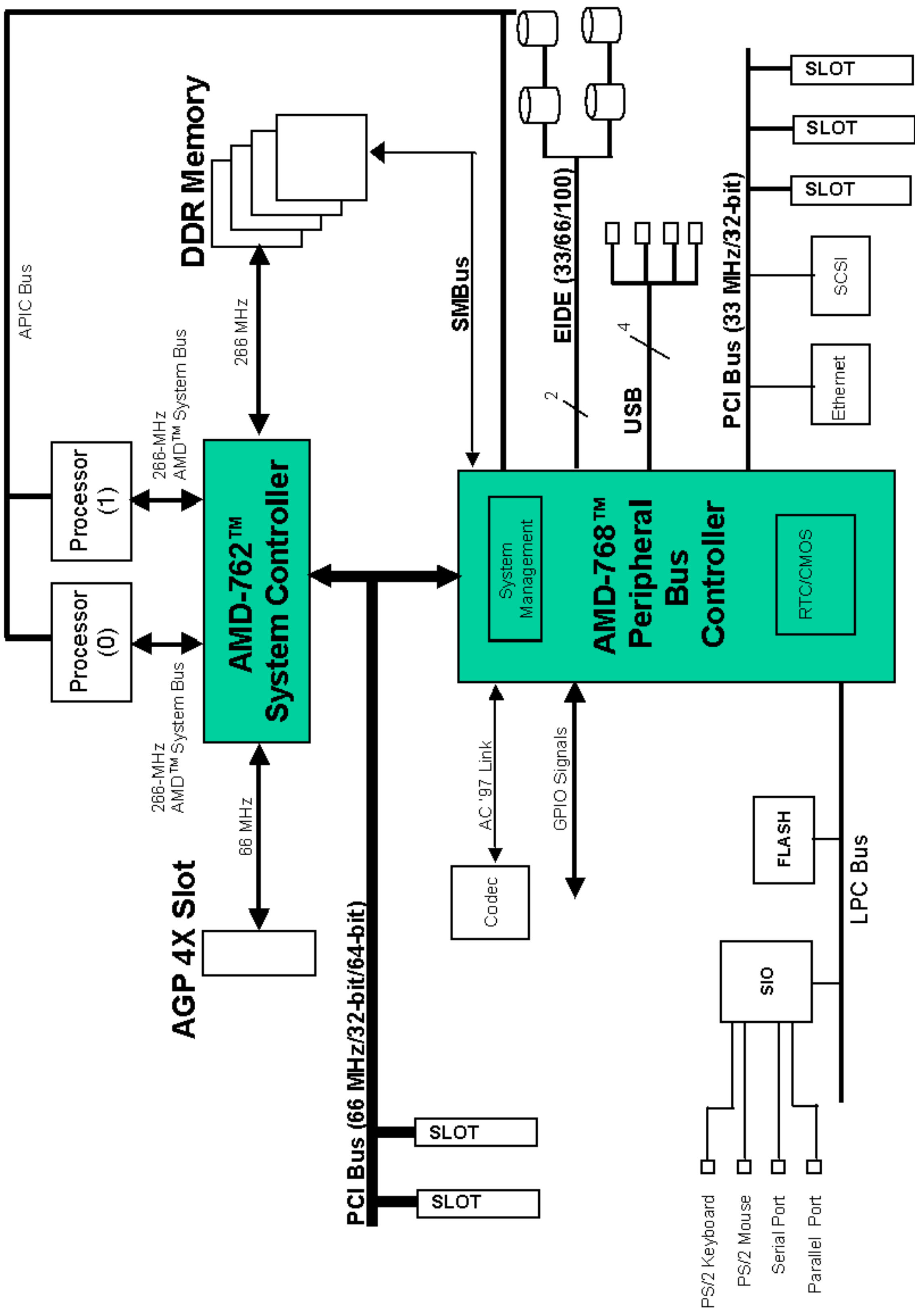

Figure 3.2: AMD-760™ MPX Chipset

3.2 Components of a personal computer

We will now examine the fundamental components of a modern PC and try to estimate their importance for a virtual computer.

3.2.1 Central processing unit

Central processing unit (CPU) or processor is the heart of every computer. This is where all the computer programs are executed and from where instructions are given to all other parts of the computer. Typical operations that a processor performs are reading and writing data from memory or a device, performing arithmetic and logical operations and reading the next program instruction from a memory location given by the results of the previous operations. In our virtual machine, we won't need to implement an actual processor that would know how to execute machine instructions for reasons given in Chapter 2, though we will still need to have some sort of control over the flow of the instruction execution. Specifically we will need to implement the interrupts feature of a processor.

Interrupts are signals that devices send to processor to inform him about a change of state of the device. For example, an input device may raise an interrupt to tell the processor that there is new data waiting in the input buffer of this device. Interrupts may also be raised by the processor itself if an arithmetic exception occurs or if the program directly requests it. Processor checks its internal interrupt flags before every executed instruction and if the flags are set, it pauses the execution of the current process and starts the interrupt service routine. The interrupts mechanism allows the processor to respond to various events when they really occur instead of having to periodically check the state of all the devices to find out if they happen to need its attention.

3.2.2 Random access memory

Memory is a device that stores all the data and machine instructions to be executed on the processor. It is principally a very simple device – it stores bytes of information at consecutive memory addresses and when requested, it drives the data on the bus or writes the data from bus to the specified memory location.

Although memory is one of the most important parts of a computer architecture, we won't really need it in our virtual machine. The reason for this is that we already have a memory given to us by the Common Language Runtime where all of our processes will execute.

3.2.3 Input and output devices

Every computer needs to communicate with the outside world so that it can receive instruction about what to do next and display results of its work. The most common type of input device is the computer keyboard and the most common type of output device is the computer screen also called the monitor.

In past these two devices used to be combined in a single device called the computer terminal. Terminals were used to access mainframe computers and usually more than one terminal was connected to a single computer, allowing multiple users to work on mainframe computer at the same time.

For the sake of simplicity, instead of implementing two devices in our virtual machine we will only implement one – a dumb terminal which will allow the user to send characters to the computer and receive characters back from computer printed on the screen.

3.2.4 Timer

An important input device is the system timer. It generates signals or “ticks” in periodic intervals and usually can be programmed to raise an interrupt after a certain number of ticks have occurred. System timer is used by an operating system to measure time and do periodic tasks such as thread scheduling.

3.2.5 PCI bus

All components of a computer are connected by devices called buses. Although modern computer buses can be very complicated, we can somewhat abstract from these concrete implementations and look at a bus as just a set of electric wires that can carry information. Buses operate in periodic cycles. In each cycle, only one device can be allowed to write data to the bus – the wires cannot carry more than one bit of information at the same time. The data travels along the whole bus so every device on the bus could potentially read this data.

One of the most widely used bus architectures is the Peripheral Component Interconnect or PCI. It is used for connecting many common computer expansion cards and numerous internal computer buses are based on the PCI specification. Three features of the PCI bus will be of particular interest to us. First one is the presence of dedicated interrupt lines. If a device needs to signal an interrupt, it can use these dedicated lines which connect directly to the PCI bridge and then they are routed to the interrupt controller. This design has a couple of flaws though from which the most important one in relation to our virtual machine implementation is that the interrupts are signaled out-of-band, meaning they are not synchronized with the bus cycles. A solution proposed in later revisions of PCI and a one that has fully replaced the dedicated interrupt lines in the PCI Express specification is to use message-signaled interrupts. In this scheme, a device signals interrupt by performing a write operation to a specific memory address. This also allows the device to attach an extra word of information to the signaled interrupt.

Second feature is the addressing mode – all devices connected to the PCI bus are at startup time assigned address spaces to be used when requesting a read or write operation. For example, a network card may indicate at startup that it needs a 1 kB block of memory for its internal buffer. The PCI bus will assign a set of 1024 consecutive memory addresses to the network card. When another device will request a read or write operation to one of these memory addresses, the address will be mapped to the network card buffer and the device will be able to read or write directly to the buffer.

The last feature we will talk about is the bus access protocol. A PCI bus includes features that allow it to control access to the bus. Since the bus can be only used by one device at a time, the PCI bus collects information before the beginning of a cycle about which devices want to access the bus. It then selects one of these devices and only this device will be allowed to perform bus operations in the next cycle.

3.3 ComOS Virtual Machine

The design of ComOS Virtual Machine incorporates all of the basic concepts we have encountered in the previous two section. Architecture of ComOS Virtual Machine is more unified than that of a common personal computer. The virtual computer contains only one bus to which all other devices are connected. In this design even the processor is treated as regular devices with almost no special status. For this reason, multiple processors can be connected to any of the device slots without causing any trouble but at the same time being fully capable of running an extra process on them.

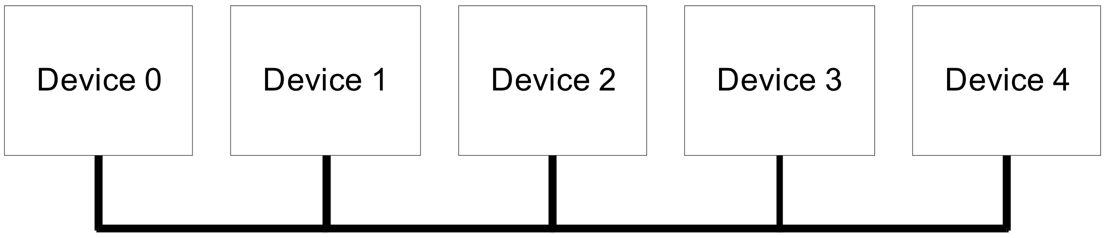

Figure 3.3: Single bus architecture of ComOS Virtual Machine

In the following sections we will go through the design and implementation of individual components and devices of the ComOS Virtual Machine as well as the machine itself.

3.3.1 Bus

All the devices in ComOS Virtual Machine are connected by a single bus. The bus is controlled by a chip called the bus controller which decides which device will be allowed to take control of the bus in the next cycle. It also assigns memory addresses to devices on the bus and resolves these addresses when devices try to read or write data to the memory. The functionality provided by the bus is reading memory, writing to memory, raising a hardware interrupt, raising a hardware exception, assigning memory resources to a device and enumerating devices connected to the bus.

Every bus cycle is divided into three phases. First is the signaling phase in which all the devices tell the bus whether they want to access the bus and what

I/O operation would they like to perform. In the second phase called arbitration, bus controller decides which device will be granted the ownership of the bus. Finally in the third phase the requested operation is performed.

The bus is capable of transferring only one byte every cycle, therefore reading and writing more than one byte will take multiple cycles. In such situations the bus controller may give ownership of the bus to one device for a longer period of time, but it will never be for more than 16 cycles to prevent monopolization of the bus. For this reason all read and write operations always return the number of bytes that was actually read or written.

Finally the bus contains dedicated channels for communicating memory mapping information. Devices can use these channels to request blocks of memory to be assigned to them as well as query the bus controller for a list of devices currently connected to the bus and the list of memory blocks assigned to these devices. Since the bus is designed with plug and play functionality in mind, a device can request a new memory mapping at any time, not only when the machine starts up. All this information is communicated by dedicated channels and out-of-bound in respect to the bus cycles, therefore no arbitration for the bus ownership needs to take place.

The bus with its controller is represented by an instance of `BusController` class which provides the following public functions:

```
public DeviceInfo[] EnumerateDevices();
public bool Interrupt(int source, int interruptInfo);
public void Nop(int source);
public void Read(int source, int address, int length,
out
     byte[] result);
public void RegisterDevice(IDevice device, int index);
public MemoryBlockInfo[] RequestIoMemory(int source,
int[]
     lengths);
public void SetInteractiveMode(int source);
public void SetPassiveMode(int source);
public void SignalException(int source, Exception ex);
public void UnregisterDevice(int index);
public void Write(int source, int address, byte[] data,
out
     int numWritten);
```

Functions RegisterDevice and UnregisterDevice are called when a device is connected or disconnected from the bus, argument index identifies which slot is the device connected or disconnected from and argument device is a reference to the object representing the device.

Functions SetInteractiveMode and SetPassiveMode switch the device connected to slot identified by argument index into interactive or passive mode. Device which is in interactive mode must signal in every cycle whether it wants to gain control of the bus by calling one of the reading or writing methods, or call function Nop to tell the bus that it doesn't want to access the bus. The BusController object waits for all devices in interactive mode to call one of the I/O functions before it ends the signaling phase of the current cycle. Devices which are in passive mode and don't call any of the I/O operations before the end of the signaling phase are treated as if they called the Nop function.

The reason for implementing the passive and interactive modes is that unlike real hardware, the ComOS virtual devices operate as C# threads and therefore are unable to call the BusController functions in precise intervals every cycle. Therefore the BusController has to wait for all the devices to signal what they want to do and only then end the signaling phase. Devices are given the option to switch into passive mode so that they won't have to call the Nop function in a loop until they need to access the bus again. All devices default to interactive mode so that they won't lose the first couple of cycles after they are connected to the bus.

Function RequestIoMemory allows the device to request blocks of memory given by the array lengths. Each element of this array is said to correspond to one function of the device. Each function will be assigned a continuous block of memory addresses. Bus controller may not be able to satisfy all the requirements due to the limited address space (31 bits) or because of memory fragmentation. The return value of the function contains information about blocks actually allocated for the device. Every call to the `RequestIoMemory` function frees all memory blocks assigned to the device by previous calls to this function.

Function `EnumerateDevices` returns an array of `DeviceInfo` objects that correspond to devices connected to the bus. `DeviceInfo` objects contain the type of the device (literally the runtime type of object that corresponds to the device) and array of memory mappings for the given device.

Function `Interrupt` tries to raise a message-signaled interrupt with `interruptInfo` as extra information carried with the interrupt. The function returns a boolean value indicating whether the bus was successfully acquired and interrupt request could be processed, or either the bus was not available or no processor was able to accept the request which means that no data was written and the device will have to try again in the next cycle. Function `SignalException` is similar to the `Interrupt` function with the one difference that it always succeeds. This is because the exception is pushed to the exception stack in the interrupt controller and forwarded to the processor the next time bus is available. `SignalException` cannot be directly called by a ComOS device. It is called automatically when an exception occurs in one of the threads simulating the device.

Finally the `Read` and `Write` functions send requests to the bus controller that a device wants to access the bus for reading or writing. Argument `source` indicates which device requested the operation and argument `address` is the memory address to or from which the operation is to be performed. In case of the `Read` function, argument `length` indicated how many bytes should be read from the destination address and argument `data` will contain the bytes that were actually read. The length of this array indicates how many bytes was the bus able to transfer. The `Write` function's argument `data` contains the array of bytes that are to be written to the destination address and the output argument `numWritten` will contain the number of bytes that were successfully transferred.

3.3.2 Interrupt controller

The interrupt controller is an integral part of bus controller. All message-signaled interrupt requests and all hardware exceptions are forwarded to it and the controller's task is to send them to the processor. Since ComOS architecture allows more than one processor in the virtual machine, interrupt controller checks all the processors in round-robin fashion and forwards any pending interrupts to processors which can currently accept new interrupt requests. Hardware interrupts that could not be processed fail to be raised. Hardware exceptions that could not be processed are kept in the exception queue and will be examined again in the next bus cycle.

ComOS interrupt controller is represented by an instance of `InterruptController` class which is a private member of the `BusController` class.

3.3.3 Device interface

ComOS bus allows a wide variety of devices to connect to the ComOS Virtual Machine. The ComOS bus is designed in such a way that all devices can be theoretically hot-plugged and hot-unplugged from the bus at any time. The unified interface through which the devices connect to the bus puts certain restrictions on them though.

All ComOS devices must be represented by instances of classes that implement the `IDevice` interface. They have to implement the following functions:

```
void NotifyConnected(Machine.BusAccessor busAccessor);
void NotifyDisconnected(Machine.BusAccesor busAccessor);
bool Read(int function, int offset, out byte data);
bool Write(int function, int offset, byte data);
```

Function `NotifyConnected` is called when the device is connected to the bus. This call is asynchronous therefore the device doesn't have to return from the call to this function in any given time interval. In particular, the device programmer could use the body of this function to execute an infinite loop in which the device will simulate its functioning. The argument busAccessor serves for communication with the bus. It provides all the functions of BusController that should be visible to the device but it will stop functioning as soon as the device is disconnected from the bus.

Function NotifyDisconnected is called asynchronously when the device is disconnected from the bus. Argument busAccessor is a reference to the same object as was passed to the corresponding NotifyConnected call.

Read and Write functions are called by the BusControler object when another device tries to read or write to a memory address assigned to this device. Argument function is the index of the function to which the requested memory address was mapped, offset is the offset within the assigned memory block and argument data contains the data to be written or serves as output argument for the read data. Return value indicates whether this device could process the call or not. If the call was not processed, the Read or Write operation fails in the same manner as if the bus was busy,

3.3.4 Processor

Although processor is just another device connected to the ComOS Virtual Machine bus, it has a certain special status. For example, the ComOS Virtual Machine won't start up if there are no processors present on the bus. This is because the ComOS Virtual Machine tries to access the first processor in the machine to give it the information necessary for booting. Also the interrupt controller treats the processor in a special way – it only forwards interrupts to processor and no other devices.

The purpose of processor is to execute code of individual processes. It contains registers which among other things store the address of the next machine instruction of the currently executing process and other contextual information such as address of the stack. Processor supports an operation called context switch in which the complete state of execution of the current process is stored in the memory and another process which was previously stored is now loaded into the processor to continue execution. Since all relevant registers are restored to the same state as they were in before this process was swapped out to the memory, the process has no way of telling that it was for some time actually suspended.

Context switch can be caused by software or by the processor itself if it encounters an interrupt. The processor contains a flag which is set whenever an interrupt occurs and all information about the interrupt is stored in the interrupt registers. As long as the interrupt flag is set, the processor cannot accept any new interrupt requests. The interrupt flag is examined after every executed instruction. If it is set then the current process is swapped out and a new thread is started which will handle the interrupt. This thread will execute the code of interrupt servicing routine, a routine whose location is stored in a specific memory address and is therefore common to all the processors in the given virtual machine. After the routine exits, the interrupt flag is cleared and execution continues from the last point before the context switch.

ComOS processor supports five types of interrupts. First two are hardware interrupts and hardware exceptions which were already explained in the section about ComOS bus. Another two are software interrupts and software exceptions – the interrupts can be raised by software by special machine instruction, the exceptions are raised automatically whenever an unhandled exception occurs in the currently executing Common Language Runtime thread. The last type of interrupt is the direct write interrupt. Every processor implements function 0 which is mapped to a memory block 1 byte long. Every write to this memory block will cause the processor to raise a direct write interrupt in case the interrupt flag is not set or fail otherwise. Direct write interrupts are the only way to signal an interrupt to a specific processor in a multiprocessor environment and are essential in operating systems supporting multiple processors.

ComOS processor is implemented as instance of the `Processor` class. As I've already mentioned in Chapter 2, ComOS processes are in fact implemented as Common Language Runtime threads and therefore are not physically executed on the ComOS processor. The processor though still provides the following functions that can be accessed by all ComOS threads:

```
public static DeviceInfo[] EnumerateDevices();
public static void Halt();
public static void Nop();
public static void RaiseInterrupt(int interruptInfo);
public static void Read(int address, int length, out
byte[]
     result);
public static Thread ScheduledThread { get; set; };
public static void SetInterruptHandler(InterruptHandler
     handler);
public static Thread StartManagedThread(MethodDelegate
     entryPoint);
public static void Write(int address, byte[] data, out
int
     numWritten);
```

All these functions are static, which means that the thread doesn't need to have a reference to any particular `Processor` object. It would be hard to keep track of it – in a multiprocessor scenario, thread could be at different times executed on different processors. For that reason the `Processor` class itself keeps track of which thread is currently executing on which processor or if it is not executing on any processor at all but is rather swapped out in the memory.

Functions `EnumerateDevices`, `Nop`, `Read` and `Write` are really calls to functions of the same name in the `BusController` class and don't need any further explanation. Function `RaiseInterrupt` has the same purpose as the `Interrupt` function in `BusController` class but with a slight difference. It doesn't cause the processor to send interrupt request to the bus but only sets its own interrupt flag, if possible. The interrupt will also be marked as software interrupt rather than hardware interrupt. If the interrupt request can't be delivered because the interrupt flag is already set, call to this method throws an exception of type `SoftwareInterruptException`.

Function `StartManagedThread` creates a new managed thread and returns a reference to it. Argument `entryPoint` is a delegate to the function this thread should execute. The call to this delegate will be decorated by an try-catch block which will raise a software exception interrupt in case the function throws an exception.

The `ScheduledThread` property can be used to examine the currently scheduled thread or to schedule a new thread for execution. If it is set outside of an interrupt servicing routine, a context switch will occur immediately. If it is accessed from an interrupt servicing routine, the value of this property is not equal to the currently running thread but rather to the thread which ran before the ISR was invoked. Setting this property will not cause a context switch until after the current ISR has exited. This property is initialized at the machine startup to an idle thread that executes the `Nop` instruction in an infinite loop. Setting this property will fail with a `ConcurrencyException` if the specified thread is already scheduled on another processor.

Interrupt servicing routines have a special behavior regarding unhandled exceptions. When an unhandled exception occurs, current routine is exited and a new software exception interrupt is raised. The `exception` parameter to this interrupt though is not the original exception but an instance of `DoubleFaultException` class. Throwing another unhandled exception from within an ISR servicing a double fault exception will cause the machine to halt. That's also what happens when the `Halt` function is called.

Calling any of these functions from a thread that was not created by the `StartManagedThread` function (and therefore is not part of the ComOS simulated software) will cause an exception. The only case when this is not so is an overloaded version of the `SetInterruptHandler` function. This can be called by the ComOS Virtual Machine at the startup to simulate the boot process – selecting the first function to execute after the machine starts.

The last function that ComOS processor offers is not part of the `Processor` class but rather a separate class by itself, the `HardwareMutex` class. Instances of this class represent words in the managed memory that can only have values 1 (locked) or 0 (unlocked). The `HardwareMutex` class provides two functions to manipulate this memory:

```
public bool Tsl();
public void Unlock();
```

Function `Unlock` only sets the value of the word to unlocked. Function `Tsl` emulates a Test and Set Lock instruction – it atomically checks the value of the word which it returns as the return value and then sets the value to locked. This instruction can be used for synchronizing processes as described in [Ham03]. This method is preferable to enabling and disabling interrupts because the latter technique doesn't work in multiprocessor configurations.

3.3.5 System timer

Every computer needs a timer to schedule certain periodic actions such as thread switching or just to keep track of time. Timer in ComOS Virtual Machine is implemented as a ComOS device in class `SystemTimer`. It's a rather simple device – in contrast to modern computer timers it cannot be programmed but only generates interrupts at a fixed rate. This rate is on average every 100 bus cycles but can be anywhere from 90 to 110 cycles as decided by the random number generator. This variation is supposed to simulate imprecisions in real computer timers, although on a very exaggerated scale. Random seed can be passed to the constructor of the `SystemTimer` device if we want the machine to behave deterministically.

ComOS timer doesn't implement any I/O functions.

3.3.6 Terminal

In this chapter I have mentioned a device called dumb terminal. It is a computer screen combined with a computer keyboard that sends encoded keyboard strokes to the computer and prints received characters on the screen. ComOS implements this device in class `DumbTerminal`. When connected to the bus, it creates a new graphical window derived from the `System.Windows.Forms.Form`. Keyboard strokes are captured in its `KeyPress` event and output is printed into a `System.Window.Forms.Label` object that fills the entire window. All characters are encoded and decoded using the UTF-8 encoding. Return key is encoded as the \n character and both \r and \n characters are treated as beginnings of a new line. This device implements the function 0 with memory block 1 byte long for both reading from the keyboard and writing to the screen. All characters pressed on the keyboard are stored in an internal buffer of unlimited length. The devices raises an interrupt every time there are new characters waiting in the buffer with the interruptInfo argument equal to the number of new bytes in the buffer.

3.3.7 Unmanaged memory

All the memory that ComOS threads need is managed by the Common Language Runtime. What would a simulated memory be good for? Well, we need to have one if we want to deal with paging and virtual memory. The ways how to exploit this device will be mentioned in the next chapter. Its implementation is rather simple – it is represented by an instance of class RandomAccessMemory and it implements single I/O function with memory block of the same size as its capacity. The content of the memory is stored in an array of bytes and reading or writing to this device modifies the corresponding entries in this array.

3.3.8 Machine

Finally, all the components we've talked about are members of the ComOS Virtual Machine implemented in class Machine. The class provides the following public functions:

```
public event MethodDelegate Started;
public event MethodDelegate Stopped;
public Processor.InterruptHandler BootHandler { get;
     set; };
public PortCollection Ports { get; };
public bool Running { get; };
public void Start();
public void Stop();
```

Functions Start and Stop do exactly what they say – they start or stop the machine. Started and Stopped are events that are asynchronously raised when the machine starts or stops, and Running is a property that indicates whether the machine currently runs or not. `BootHandler` property gets or sets the value of the machine's boot handler. This is the delegate which will be set as machine's interrupt handler when the machine starts. Property `Ports` returns the collection of ports to which devices can be connected or disconnected from. These devices are in fact conected to the internal bus but we don't want to expose the reference to the `BusController` object itself.

Chapter 4

ComOS software

ComOS Virtual Machine is a complete computer simulation capable of running native Common Language Runtime threads. Well, actually we should really say “running”. Software that runs on the ComOS Virtual Machine is in fact executing in the CLR as we saw in Figure 2.1 and only has the convenience of also being able to call the static methods of the `Processor` class. How do we convince it to use these methods? How are we going to enforce the restrictions we defined in Chapter 2?

4.1 Isolating the software from CLR

Surprisingly there are quite a few possibilities to achieve this. All of them take advantage of the fact that we have full control over how the ComOS software will be compiled – remember that the primary use of ComOS is to have students program the ComOS Virtual Machine. Why not have them submit their source code and then compile it at our machine?

Let's state again the problems we would like to solve:

- ComOS software should not access certain features of the Common Language Runtime such as reflections, threading and synchronization primitives.
- Threads simulated on the ComOS Virtual Machine should only execute when they are scheduled on the processor. We also need a mechanism for interrupting currently running threads. In particular, inserting a call to `Processor.Nop` after every MSIL instruction of ComOS software would solve these problems.

We will now explore the options we have and assess their usefulness. Unfortunately most of them will turn out to be very complicated and impractical.

4.1.1 Ignoring the problem

Now this option is really not very complicated at all but it also doesn't solve any of our problems. A slight modification of this approach would be telling the ComOS programmers to follow the given rules and then check that their source code complies with them. Sounds easy, but do we really want every second line of their source code to be a call to `Processor.Nop`? Probably no. We will only resort to this if we can't find any other reasonable approach.

4.1.2 Code processing

If we'll think about the previous proposed solution a little more, we will realize that it could be automated. What we need to do is to write a program that will receive a C# source code as an input, inspect it and tell us if it contains any restricted calls or language constructs. It will then insert the `Processor.Nop` calls to all places in the source code where function calls are permitted, such as function bodies. It would be very easy to incorporate this program into compilation with Microsoft Visual Studio pre-build events.

Sounds easy? It sounded doable to me, until I saw the C# Language Specification [Ms07]. It is 493 pages long. Of course you could wonder if there are possibly some existing C# code processors. Maybe there are, but none that I know of. And frankly, this approach can't really appeal to anyone.

4.1.3 Debugger API

If you've ever worked with one of the Microsoft Visual Studio IDEs you may have wondered how it allows you to step your program line by line. You can now find out by yourself because Microsoft made the debugger API that Visual Studio (perhaps) uses publicly available. Unfortunately, it is not easy to learn and there exists only a poorly documented semi-functional C# version of it. (The debugger API is for the C++ language.) I haven't even been able to find out what functionality does it exactly provide so I can't honestly tell if mastering this API would solve our problems.

4.1.4 Post-compilation

The best option by far turned out to be post-compiling our code. Post-compilation means editing the binary after it's compiled into MSIL code. It's pretty much the same approach as code processing but in MSIL instead of C#. How could that be better? In fact, the MSIL language is much simpler than C#! And for this reason, there exist many nice and freely available post-compilation libraries. The one I've chosen for ComOS is PostSharp, an aspect weaver for the .NET platform. It's greatest advantage is that it integrates into the Microsoft Visual Studio and allows you to control the post-compilation directly in your C# code. All that needs to be done is to write a custom attribute which will be then applied to the target assembly, in our case the assembly `Comos.Software`. This will tell the PostSharp platform to post-compile the assembly according to the rules defined in our custom attribute.

So what will PostSharp allow us to do? First of all, it allows us to issue compile time warnings or errors if it finds calls to restricted functions anywhere in the post-compiled code. This solves problem number one. Secondly, we can use PostSharp to transform certain parts of MSIL code which in our case means adding a call to `Processor.Nop` after these parts. What parts are we talking about? PostSharp allows us to modify all of the following:

- function calls
- constructor calls
- reading an array element
- writing an array element
- reading a field
- writing a field
- getting a pointer to a field

That's a lot of possibilities but these certainly aren't *all* of the MSIL instruction. However, what are the real reasons why we want to insert the `Nop` calls? First of all, we are only able to interrupt the thread and start simulating the next clock cycle when the thread calls the `Nop` function (or one of the other I/O functions). Inserting the `Nop` call after every MSIL instruction would mean that one clock cycle of our processor would correspond to one executed MSIL instruction. But since the options PostSharp gives us cover almost all of the common instruction, we wouldn't really lose that much. A code like this:

```
while (true) { /* do nothing */ };
```

would still put our processor in an infinite loop but as soon as there's a single call to any function whatsoever or access of any class field or even local array, we're safe again.

A more important reason why we want to periodically interrupt our code is to force the ComOS software programmers to synchronize their code. Consider the following code:

```
if (this.obj != null) {
      this.obj.DoSomething();
}
```

If the field `obj` was shared among multiple threads, the previous block of code would have to be synchronized to avoid a situation where object `obj` is a non-null reference during the evaluation of the if condition but becomes a null reference right before the `DoSomething` method is called, this resulting in a `NullReferenceException`. However, if the ComOS programmer knew that the thread cannot be interrupted between these two calls, he wouldn't have to synchronize anything.

What I will try to do now is to examine situations where such synchronization issues could occur. First of all, there needs to be some shared resource in use by more than one thread. This resource must be either an object to which we have a reference or a variable that holds a certain value. (Note that the above example classifies as a value and not an object, because we are sharing a reference to some object (i.e. its address in memory), not that object itself.) In the case of object, all is well. This is because the only way to access an object is to either call one of its functions or access its public fields, both of which are cases where the post-compilation occurs. There is one special kind of an object though – an array. Arrays can be accessed in a third way, by reading or writing their elements. Luckily for us, this again is one of the post-compiled case.

Now let's consider variables. If the variables are fields of a class or structure, we are again covered by PostSharp. We can do nothing if they are local variables but fortunately, there is no way that local variables could be shared among threads because they only exists during the scope of the current block. (Static variable don't exist in C# or MSIL.) There is one tricky situation we haven't thought of though – pointers to variables. How can you have a pointer to local variable? Easily:

```
int localVariable;
obj.Function(ref localVariable);
```

When variables are passed by reference, only pointers to them are passed to the function. If the target function was executed on a new thread, variable `localVariable` would become a shared resource. PostSharp allows us to post-compile the piece of code where the reference is created, but it won't allow us to control any further access to this reference. For this reason, we will not allow to pass variables to functions by reference but generate a compile time error instead.

Of course, there are many other cases we haven't thought of where PostSharp cannot handle the threat of shared variables such as anonymous delegates that use local variables. We will have to live with the fact that our post-compilation mechanism is not perfect.

4.2 Using the ComOS project

So what is it that we can actually do with ComOS? The purpose of the ComOS project is to allow students to write their own operating system in the C# language. This is quite possible but unfortunately also a bit complicated task to accomplish. ComOS comes with a simple software demo which includes a driver for the `DumbTerminal` device and an implementation of an interrupt servicing routine that prints any software exceptions to the screen. There is clearly room for improvement – a basic operating system framework needs to be set up so that students will only need to implement small parts of code. It would also be nice if the machine was in future capable of executing unmanaged code – for example programs compiled for some simple RISC processor such as MIPS. This would allow students to use the `RandomAccessMemory` device for some useful purpose and see how virtual memory really works. Adding a hard drive device would then also be necessary.

What the project can readily be used for is the following:

- writing implementations of basic synchronization mechanisms such as semaphores and locks
- writing methods for communication between threads and thread scheduling such as join, sleep, wait and pulse methods
- designing a thread scheduler

ComOS Virtual Machine can also be used in multiprocessor setup to experiment with multiprocessor thread schedulers. It is not entirely clear though whether the direct write interrupt mechanism is sufficient to allow an efficient implementation of multiprocessor operating system.

The modular architecture of ComOS Virtual Machine is quite versatile. ComOS Virtual Machine could be possibly used for any of the following educational purposes:

- implementing an operating system (the original goal)
- designing and testing efficient parallel algorithms on a multiprocessor ComOS Virtual Machine
- implementing a network protocol or network card driver and simulating a computer network using multiple instances of the `Machine` class
- implementing distributed algorithms in an environment with multiple networked ComOS Virtual Machines

Chapter 5

Conclusion

In this paper I have studied the architecture of modern computers and proposed a model of a virtual computer that would closely simulate the most important parts of a real-world computer. I have implemented this virtual computer in the C# language and I have also proposed a use of this computer as an educational tool in the 1-INF-170 Operating Systems course.

Although the virtual computer still lacks software that will be needed if we were to put it in real use in the classes, writing the software is only a minor task in comparison to the hardware implementation of the virtual machine. The ComOS project can already be used to demonstrate parts of the curriculum of the Operating Systems course and it requires only relatively small modifications to be used for demonstrating most of the concepts. Though only time will show how successful will it be.

List of figures

Bibliography

[AW02] Archer, T., Whitechapel, A., *Inside C#, Second Edition*, Microsoft Press, 2002

[Ham03] Hambálková, V., *Operačné systémy*, Comenius University, 2003

[Sin] Microsoft Research Singularity Project
`http://research.microsoft.com/os/Singularity/`

[HC01] Hettena, D., Cox, R., *A guide to Nachos 5.0j*, 2001
`http://www.cs.berkeley.edu/~kubitron/courses/cs162-F05/Nachos/walk/walk.html`

[AMD01] *AMD-760™MPX Chipset Overview*, Advanced Micro Devices, Inc., 2001
`http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24494.pdf`

[MS07] *C# Language Specification*, Ver. 3.0, Microsoft Corporation, 2007
`http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc`

[Fra08] Fraiteur, G., *PostSharp 1.0 User Guide*, 2008
`http://doc.postsharp.org/1.0/`

Abstrakt

Virtuálny operačný systém je operačný systém ktorý miesto skutočného počítača beží na simulovanom virtuálnom počítači. Môže byť použitý na výuku ale taktiež môže mať uplatnenie aj v bežnom svete. Výhodou virtuálnych operačných systémov je zvýšená bezpečnosť, ktorá vyplýva z toho, že programy bežiace na virtuálnom operačnom systéme sú striktne oddelené od zvyšku počítača. Virtuálne počítače majú tiež často dostatočne jednoduchú štruktúru na to, aby nám dovolili uskutočniť dôkaz korektnosti programov, ktoré na nich bežia.

V tejto práci predstavím virtuálny operačný systém Comenius (ComOS) a virtuálny počítač na ktorom pracuje – ComOS Virtual Machine. Operačný systém ako aj virtuálny stroj ComOS sú napísané v jazyku C# a postkompilované knižnicami PostSharp, čím spolu tak tvoria jednotný projekt. Cieľom tohto projektu je vytvoriť prostredie, v ktorom budú môcť študenti operačných systémov navrhovať časti svojho vlastného operačného systému a riešiť problémy s ktorými sa bežne stretávajú programátori skutočných operačných systémov.

Kľúčové slová: ComOS, virtuálny operačný systém, PostSharp