Discussion of virtual machines with the concept of abstraction. Weshould all be familiar with the idea of abstraction – it is fundamental to computer science.

If you've ever used a subroutine in your code, then you've used abstraction to solve a problem.

The idea is that designing modern computer systems is very difficult. So, the design process itself is partitioned into several hierarchical levels.

These levels of abstraction allow implementation details at lower levels of a design to be ignored or simplified, thereby simplifying the design of components at higher levels.

A quote about abstraction by David Wheeler, who was an early computer scientist, he actually received the first-ever PhD awarded in computer science in 1951. He also is credited with inventing the subroutine. He said "Any problem in CS can be solved with another layer of indirection – except of course the problem of toomany layers of indirection"

The book gives an example of a hard disk that is divided into tracks andsectors in hardware.

The hard disk interface is abstracted away by the operating system so that the disk appears to application software as a set of variable sized files.

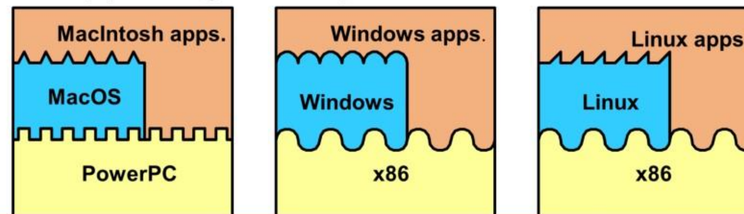
So, your system software: compilers, operating system, and middleware, are all designed to provide abstraction.

And the lower levels of abstraction are typically implemented in hardware, with real physical properties.

In the upper-level software, we aim to provide components are logical so that people can use them without worrying about their physical properties.

Interfaces

- Define the communication b/w two entities
  - Hierarchical relationship
  - Linear relationship
- Software can run on any machine supporting a compatible interface



THE UNIVERSITY OF TENNESSEE KNOXVILLE

The other aspect of managing complexity is the use of well-defined interfaces.

An interface is a shared boundary that two entities can use to communicate.

This division between the entities can allow either a hierarchical relationship between components (mainly to manage complexity), or a linear division to allow the development of components in parallel.

The instruction set is an example of an interface.

Hardware designers at Intel develop microprocessors that implement the x86 instruction set. And software engineers at Microsoft can write compilers that map their high-level language to the same instruction set. Since both use the same interface, the compiled software will execute on the microprocessor.

The idea is illustrated in the figure here. Macintosh apps use the interface provided by the MacOS to access system services. So, the interface here might be the system calls and other facilities provided by the MacOS platform. This operating system and

application software would be compiled into PowerPC instructions that execute on a PowerPC processor.

Similarly, the windows applications and linux applications might all be compiled to run an x86 machine. Because they use the same interface, they can run on the same hardware.

There are several advantages to using well-defined interfaces

Interfaces allow computer design tasks to be decoupled, so that teams of hardware and software designers can work more or less independently.

Another advantage is that interfaces can help manage system complexity. Developers use interfaces to hide the design details of each component by providing an abstraction to the outside world.

For example, application software does not need to be aware of changes that occur inside the operating system, so long as the interface to each service remains the same.

This makes it easier to upgrade different components on different schedules.

There are, however, a few disadvantages to using interfaces.

For one, they can limit flexibility. Forcing developers to always use the same interface, even if its not the best way to do their task, will result in sub-optimal design.

Well-defined interfaces can also be confining since components designed to specifications for one interface will not work with those designed for another.

For instance, an operating system designed and compiled for a particular ISA will only work on microarchitectures that implement that interface.

Similarly, application binaries are tied to a particular instruction set and operating system.

So, you can't run ARM binaries on an x86 machine.

What about Windows applications compiled to x86? Can we run a Windows application compiled to x86 on an x86 machine running Linux? Why not? Different system calls and libraries.

So, as a general rule diversity in operating systems and instruction sets is a good thing because it encourages innovation and finding new and better ways to do things. But we do not see as much diversity across these systems as you might think because it is very hard to change an instruction set or operating system once it becomes widely used and a lot of upper-level software depends on it.

For instance, think about what was the most recent successful new ISA or OS? ARM? How old is ARM now? And ARM only emerged in the embedded domain where there wasn't a lot of legacy code that needed backwards compatibility.

For everything else, people still mostly use x86 – and (and I say this as someone who has worked at Intel and really likes Intel) x86 is not a particularly good instruction set architecture – especially now that more recently the focus has turned to power and efficiency over performance. However, attempts to replace x86 have mostly failed due to the need to provide backward compatibility to the x86 interface.

Another disadvantage is that application software may not be able to directly exploit features in the microarchitecture. Now, this isn't really a disadvantage because

application software is supposed to be architecture independent. For instance, you cannot write C code that directly manipulates which values go into registers and which will go to memory to improve performance. For this purpose, we rely on compiler and runtime software to exploit low-level features, while still allowing upper-level software to remain portable.

Virtualization is technology that you can use to create virtual representations of servers, storage, networks, and other physical machines.

Now, we can overcome some of the disadvantages of interfaces by using virtualization.

When a system (or subsystem), is virtualized, its interface and all resources visible through the interface are mapped onto the interface and resources of a real system actually implementing it. Consequently, the real system is changed – in that it now appears to be a different, virtual system.

Cell phone example

So, the idea is that you take a system, for instance, let's say your cell phone. And you want to run the applications that run on your cell phone on your laptop. The way you could do this is to virtualize the cell phone system using your laptop as a host. This means you write software that implements the cell phone interface and install it on your laptop. Now, you can magically run the cell phone applications on your laptop! In this example, the cell phone's interface is called the guest system and the laptop platform is called the host.
Virtualization provides a way to overcome some of the constraints imposed by interfaces, and to increase the flexibility of our systems.

For one, it improves the availability of application software because it allows us to run the software anywhere a virtual machine is running.

Additionally, it can improve security and failure isolation. In most systems, there is an implicit assumption that the hardware resources of a system are managed by a single operating system, under a single management regime. However, if we run each application in its own virtual machines, it's easier to isolate them from the other users – which can improve security and make it easier to isolate failures.

One other thing to note about virtualization is that, while it is similar to abstraction in that it is providing an interface to a resource, it's different in it's goal. The aim of virtualization is not necessarily to provide a simpler view of the interface. Rather, the main goal of virtualization to increase flexibility by providing other systems access to the interface.

Formally, virtualization constructs an isomorphism that maps a virtual guest system to a real host.

The function V represents the mapping of state from guest to host. The state of a system might be whatever is in its registersand other storage devices.

And for some sequence of operations e, that modify the guest state, we create a corresponding sequence of operations called e' in the host that can perform an equivalent modification on the state in the host machine.

While the isomorphism on the previous slide could also be used to describe abstraction, virtualization is different because it does not necessarily try to hide the details of the original system in the virtual system. Indeed, the level of detail in a virtual system is often the same as the real system.

Consider this example with hard disks. If you wanted to write software that virtualizes the operation of a hard disk – here's one way you could do it.

You create virtual disks that are mapped to a real disk by implementing each of the virtual disks as a single large file on the real disk. Then, you have a layer of virtualization software that implements the hardware interface of the disk using the file. The level of detail provided by the virtual disk interface, i.e. sector and track addressing, remains the same as the real disk – and so no abstraction has taken place.

We have seen what virtualization is, and how it is different from abstraction.

Now, applying the concept of virtualization to virtualize the interface for an entire machine is called a virtual machine.

A virtual machine adds a layer of software to a real machine to support whatever architecture you want to use.

For example, if you want to run Windows applications on a Mac, you could build a virtual machine that implements the interface of windows on the Mac system.

The process of virtualization consists of two parts:

1) it maps the virtual resources or state of the machine you want to virtualize to real resources. So, if the machine you want to virtualize has some data stored in memory – you would have to store this data in memory on the real machine to virtualize it.

2) It must use real machine instructions to carry out the actions specified on the VM. So, for each instruction the VM might get, you need a corresponding implementation of that instruction on the real machine

There are a wide variety of virtual machines that are used and these machines provide a wide variety of benefits. So, let's talk about some of the benefits of virtual machines to give you an idea of where we're going.

For one, they increase flexibility. Since multiple VM's can run simultaneously on the same system, we can use VM's to provide users whatever operating system or platform they want to use.

In the same vein, they increase portability. Java applications can run across a network of machines with different instruction set architectures and operating systems because they run in a virtual machine environment.

Another benefit is isolation – processes running in one VM are not allowed to affect or interact with other processes on the system – the application is said to run in a sandbox.

Isolation is important for security because a malicious application cannot affect other process, outside its own VM. Without this layer of security, it would not be possible to run untrusted internet applications on our desktops.

This is not a complete list – we'll see more uses later – I just wanted to give you some idea of why virtual machines are useful and important.

A study of virtual machines is also a study of computer architecture.

VM's are often used to bridge architectural boundaries, and a major consideration in constructing a VM is the accuracy with which a VM can implement the architected interface.

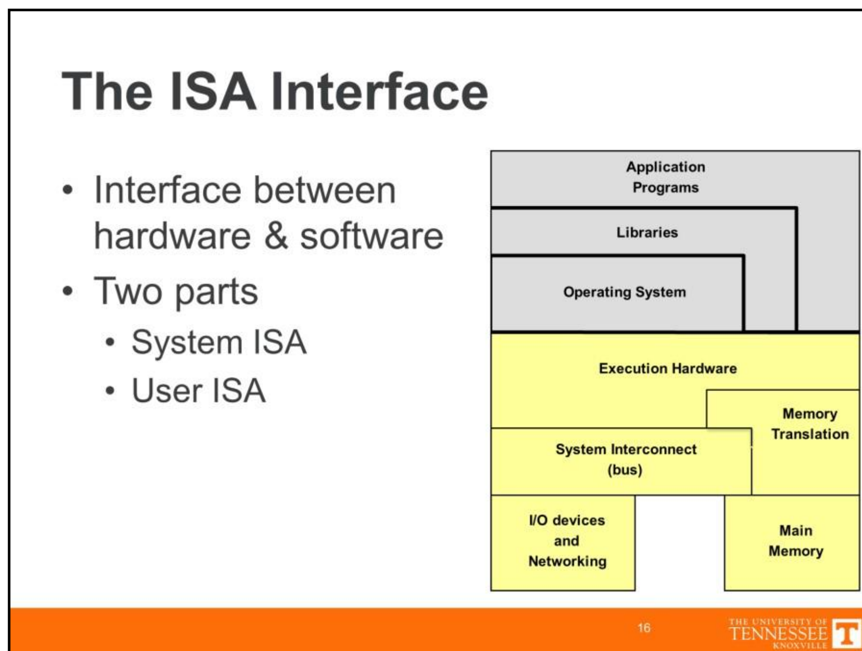
So, for our study, it is useful to define and summarize what is meant by computer architecture.

The architecture of a computer system refers to its functionality and appearance to its users, but not the implementation details. You can describe an architecture through a specification – you don't actually need an implementation.

The implementation is the embodiment of the architecture – you can have several implementations for a single architecture (for instance, a low-power implementation, or high-performance implementation, etc.)

So, let's talk about the architecture of a computer system.

Computer systems are built from layers of abstraction and well-defined interfaces. Let's look at some of the important interfaces here.



The instruction set architecture is the interface to the execution hardware. It's shown

in the middle line in the figure – it marks the division between hardware and software.

There are two parts of the ISA that are important for virtual machines: the system ISA and the user ISA

The system ISA is the part of the ISA that is only available to supervisor software – such as the operating system.

The user ISA consists of all the instructions that are available to all the user-level software.

The system ISA is really only important for OS developers who need access to these privileged instructions.

The user ISA is part of what is known as the Application Binary Interface (or ABI).

The ABI consists of the user ISA and system calls for the operating system.

So, the ABI provides your user-level programs with access to hardware resources and services. If your applications need access to some privileged instructions, they can issue instructions from the system ISA by calling system calls.

The ABI interface is important for user-level code that needs direct access to system resources and instructions – so, for instance, compiler writers will make extensive use of the ABI.

There's a couple other important points I want to make here:
The ABI only includes the user-level ISA – it does not include instructions that are in the system ISA.

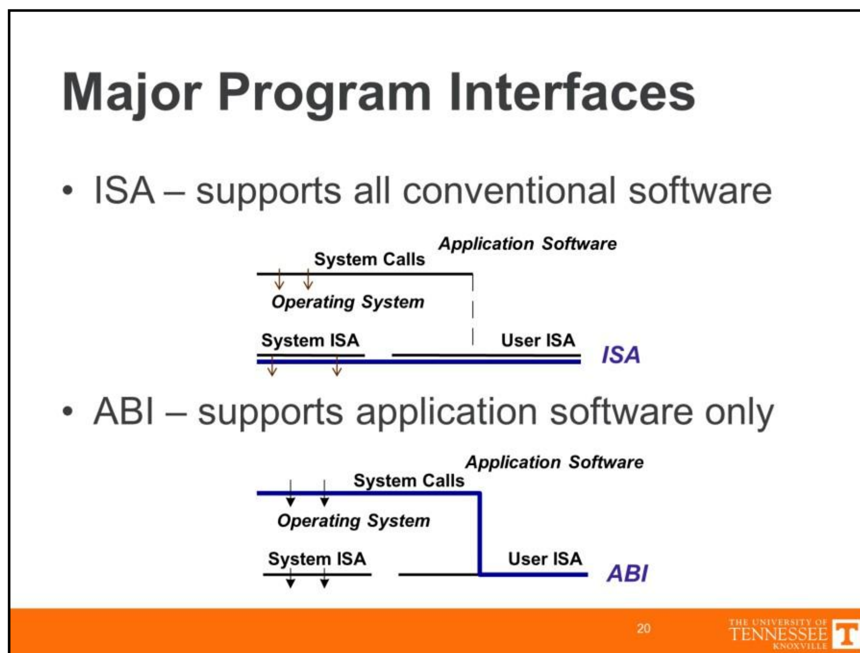
Also, any program that is written for a specific ABI can run unmodified on a system with the same operating system and ISA.

Another important interface is the application programming interface (or API). The API consists of the user-level ISA plus library calls.

API's are typically defined with respect to some high-level language using a standard library. In this way, applications written in the API can be ported easily. For example, the Python Standard Library is an example of an API.

Library calls might rely on lower-level system calls and other parts of the ISA to achieve the functionality the application requires.

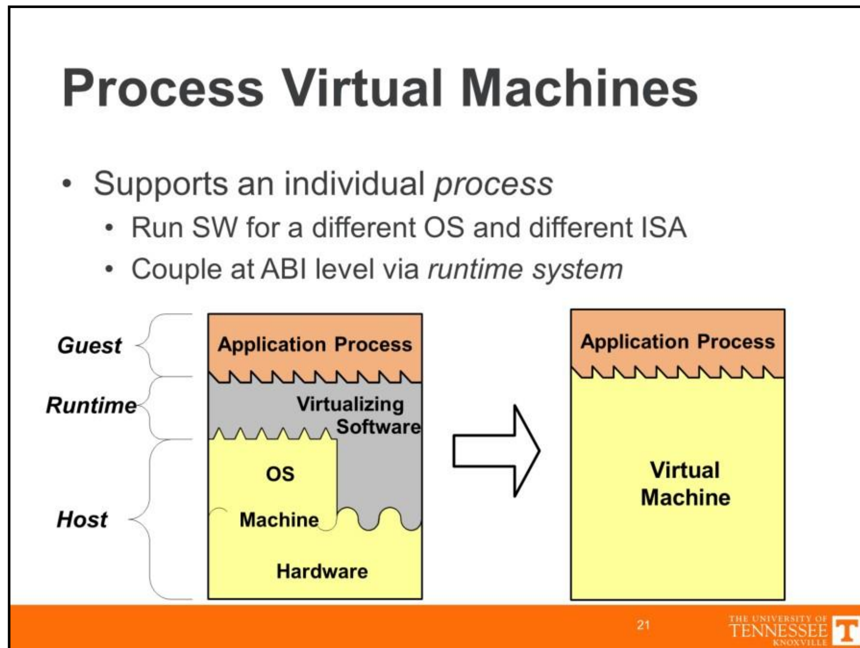
One thing to note here is that this figure is not really accurate for C and C++ programs. C applications can call system calls directly – many high-level languages do not allow applications to directly call system calls – as shown in the figure.



In order to understand the different types of virtual machines, let's first consider what the meaning of a "machine" is from different perspectives.

From the perspective of the operating system, an entire system is supported by the machine. The machine supports a full execution environment that is multi-process and multi-user, and allows access to the I/O resources. The system environment is not transient – that is – it persists over time. Hence, the machine, from the perspective of the OS is implemented by the underlying hardware – and the system interacts with that machine via the ISA.

From the perspective of a process executing a user program, however, the machine consists of a logical memory address space, user-level registers, and user-level instructions. The I/O part of the machine and the privileged instructions are only visible through the OS system calls. Processes are typically transient – they start, do some work, and then die when they're done. From this perspective, the machine is a combination of the operating system and the underlying user-level hardware (or the ABI).



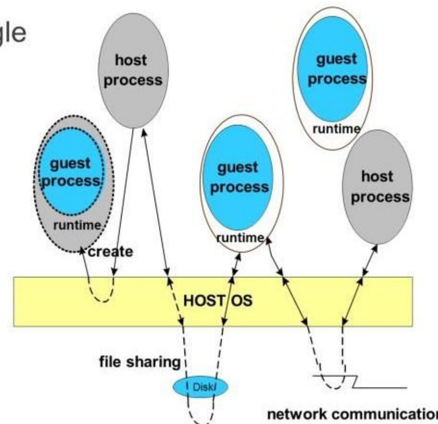
Now, I want to talk about some of the types of virtual machines.

A process virtual machine is capable of supporting an individual process. It virtualizes the resources necessary for the process to run on a different machine. In this way, you can run software written for a different operating system and different user-level ISA on whatever real machine that you use to run your virtual machine.

In process VM's, the virtualizing software itself is often called the runtime or runtime system. So, I'll use this term sometimes when talking about a process VM – I hope this is not too confusing considering the broader definition we discussed last time.

Process Virtual Machines

- Guest processes intermingle with host processes
- Binaries encapsulated by the runtime
- PVM does not include OS
- Examples:
  - Java
  - IA-32 EL
  - Dynamic optimizers (dynamo)



THE UNIVERSITY OF TENNESSEE KNOXVILLE

This figure shows a typical environment for programs running in process virtual machines.

By using a process VM, the guest program, which was developed for another system, can be installed and used in the same way as all other programs on the host system.

The user as well as the other programs interact with the guest programs in the same way as they interact with native host programs. And additionally, the guest processes can interact with each other just as if they were running on a real machine.

A common example of a process VM would be something like the Java virtual machine. For Java programs, you compile the high-level source code to bytecodes, which are binary instructions specifically designed to execute on a virtual machine.

The binary instructions are encapsulated by the runtime system and execute inside the runtime environment.

Notice that the runtime system does not include an operating system. To access system services, it will use the interface provided by the host OS.

I've also got some other examples listed here on the slide – we'll talk about these in just a moment.

PVM: Multiprogramming

- OS provides PVMs for each application
- System calls + user ISA == PVM to execute multiple, concurrent processes
- Each process is given the illusion of having the entire machine to itself

23

In the text, the authors next provide a survey of the various ways process VMs are used in real systems.

One use case is so ubiquitous that we don't often think of it as using a virtual machine – but the process construct that we use in multi-process operating systems is a form of VM.

The combination of the OS system call interface and the user instruction set form the machine that can support multiple, concurrent user processes.

Each process is given its own address space and access to a file structure. In this way, it appears to the process that it has the machine all to itself.

PVM: Emulators

- Execute binaries compiled for one ISA on a machine with a different ISA
- Emulation methods
  - Interpretation
    - Fast startup, but slow steady-state
  - Dynamic binary translation
    - High startup overhead, faster steady-state
    - Uses a code cache to store translated blocks
- Examples: Java, IA32-EL

24

A more challenging problem for process virtual machines is to support program binaries compiled to a different instruction set than the one executed by the host's hardware, i.e., to emulate one instruction set on hardware designed for another.

The most straight-forward method of emulation is interpretation.

Interpretation fetches, decodes and emulates the execution of each individual target instruction on the host machine. This process is slow, but there's really no overhead to start the interpreting the instructions.

To speed up emulation, process VM's will often use a technique known as dynamic binary translation. This process transforms blocks of source instructions into equivalent instructions on the target platform.

This will, obviously, take longer to start up because the VM has to do this transformation before it can use the code, but it typically runs much faster than interpretation once a significant portion of the program is translated.

To maximize reuse, the compiled instructions are typically stored in an area of

memory called the code cache. We'll discuss code caches later in this course.

Because of this different performance characteristics, many systems will use a combination of both interpretation and DBT to get good startup performance and good steady-state performance.

So, they start by interpreting the guest application's source instructions, and use profiling to determine which sections of the program are executed most frequently. Then, the hot sections of the code are compiled and may be optimized.

There are many examples of VM's that do emulation. I've mentioned Java. Another is the IA-32 EL (or execution layer) VM which was developed by Intel.
Intel originally developed the IA-32 instruction set, which is the 32-bit version of their x86 instruction set and it was relatively successful. Later, Intel developed a newer 64-bit ISA, called IPF, which was used in their Itaniumfamily of processors.
But now, programs written for IA-32 would not run on their Itanium processors. So, they developed IA-32 EL VM, which enables applications written in IA-32 to run in an IA-32/Windows environment on their Itanium processors.

In some systems, they might use a process VM even though the source and target instruction sets are the same. In this case, the purpose is not for portability, but to optimize the code in some way.

This can be very useful for source binaries that, for whatever reason, were not optimized. Binaries are often shipped without optimizations, due to lack of development time, or problems with the optimizer, since optimized code needs to be tested again, and it is difficult to debug.

There are tradeoffs to doing optimization at runtime. One downside is that the optimizer loses some of the high-level program information, such as data types, or method information. But an advantage is that we can collect profiling information about how the application behaves at runtime. This can help guide our optimization decisions.

An example of a dynamic binary optimization framework is dynamo, which was developed by HP in the late 1990's, early 2000's. I don't think it is used much today, but it was the precursor to several binary translation tools, such as DynamoRIO, Pin, and valgrind, which are used extensively as profiling and debugging tools.

So, for portable execution, it is possible to virtualize existing ISA's (such as x86). However, since the real-world ISA was not designed to be virtualized, this process is complicated, and exact virtualization may be slow, difficult, and, in some case, not actually possible.

High-level languages were designed with the goal of being virtualized to enhance portability.

If you design a language from scratch knowing that you want to execute its instructions in a virtual environment, you can simplify the virtualization process by choosing language features that are as independent from the hardware and operating system as possible.

For instance, the code you distribute might use an execution model based on a stack rather than a machine with registers to remove any assumptions about how many registers the target machine is likely to have. Similarly, rather than relying on system calls from a particular OS, applications are written using a library of functions that need to be ported to each platform.

The figure on this slide shows how code distribution for VM-based languages differs from traditional programming languages, such as C.

In conventional systems, a program written in a high level language is compiled using a compiler that translates the source code to an intermediate form. This intermediate code is then converted to platform-specific object code before distribution.

In HLLs designed for VMs, the HLL source code is similarly converted into intermediate code, and it is this intermediate code that is distributed. However, the intermediate code is designed so that later interpretation or translation by a VM is simplified.

At the same time, using a VM allows for a number of benefits other than just portability, such as dynamic loading of libraries, and garbage collection.

So, in Java, you use the javac compiler to create a class file. This class file contains the bytecodes that make up a Java program.(example on command line?)

The bytecodes in Java are a virtual instruction set architecture. They are just like any other ISA – but there is no actual hardware that implements this ISA. The thing that will actually execute the instructions in the Java class files is the Java virtual machine itself.

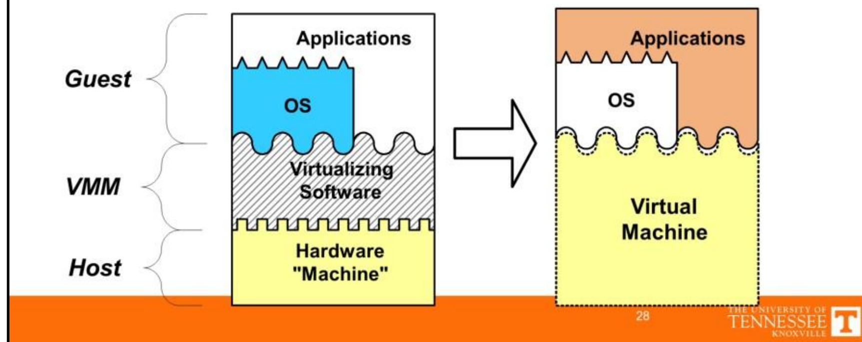
There have been implementations of the JVM ISA in a few embedded processors for maximum performance, but they are certainly not in wide use.

There are no system calls, only library calls that are handled bythe runtime.

In addition to Java, the common language infrastructure by Microsoft is another example of a high-level language VM. The .NET framework is an implementation of the CLI.

System Virtual Machines

- Supports an OS with many user-level processes
  - Couple at the ISA level
  - Examples: VMWare, Transmeta Crusoe



Another type of virtual machine is the system virtual machine.

A system VM provides a complete system environment. It provides a guest OS, with all OS-level accesses, such as to hardware resources, I/O, display, etc. it will support the OS as long as the system environment is alive.

For a system VM, the virtualization software is placed between the underlying hardware machine and the conventional software. The VM emulates the guest's ISA. In this way, the guest software sees a different ISA than the one supported by the host.

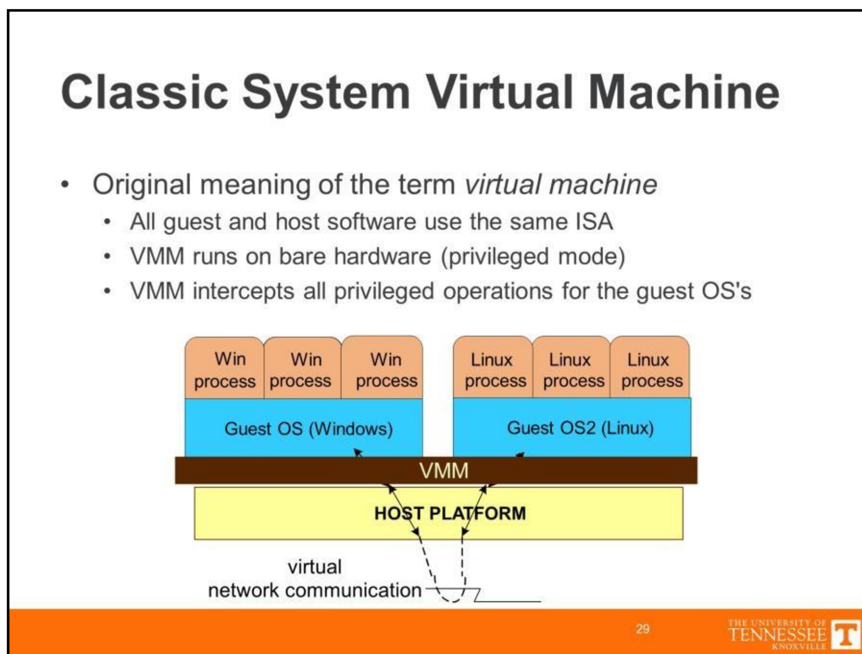
Here, the virtualizing software is referred to as a VMM or Virtual Machine Monitor. It's also called a hypervisor. Notice here, the VMM runs under the system's operating system.

I'll talk about these examples over the next few slides.

One example of a system virtual machine would be like VMWare. They're probably the biggest company that provides virtualization solutions for enterprise software.

Another example the authors discuss in the book is Transmeta Crusoe. Transmeta built low-power processors that used a VLIW architecture. Then, they used a VMM to run software compiled for x86 on these low-power devices. Using this combination, they were able to achieve similar performance as Intel's hardware with less power cost.

Recently, they haven't sold much hardware. They just sell their IP.



Let's talk about a few different types of system virtual machines.

First is the classic system virtual machine. These classic virtual machines basically partition the hardware resources among multiple OS environments. Indeed, this was the original meaning of the term virtual machine. Today, in addition to multiplexing hardware resources, these VMs are used to provide isolation and security to each user.

So, in this configuration, the VMM goes on bare hardware and the guest virtual machines go on top of the VMM, and all the guest software uses the same ISA as the host platform. The VMM runs in privileged mode, while all the guests systems run with lesser privileges. In this way, the VMM can intercept all the guest OS's actions that need to interact with hardware resources.

This implementation is very efficient – it can provide service to all the guest VM's in a more or less equivalent way.
One disadvantage of this approach is that, to install the VMM, you need to completely wipe the existing system clean and start from scratch. Also, the VMM needs to implement device drivers for your I/O devices because only the VMM is going to interact directly with the hardware.

Hosted VM

- VMM built on top of existing OS
- Advantages
  - Installs just like a user-level application
  - Host OS provides driver support
- Drawbacks
  - Less efficient

30

An alternative VMM implementation is to build the virtualization software on top of an existing OS. This is called a hosted VM.

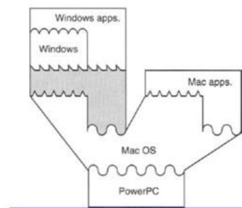
This is the approach of some popular virtualization software (such as Oracle's VirtualBox, or qemu).

The advantage of a hosted VM is that it installs just like any other user-level application – you don't have to wipe your machine to use it. Furthermore, the VMM can rely on the host OS to provide device drivers and other low-level system services.

The disadvantage, of course, is that this approach is less efficient than the classical approach because you have to go through more layers of software when you need to access an OS-level service.

Whole System VMs

- Host and guest do not use a common ISA
  - Both app and OS code require emulation
- Typically implemented as a hosted VMM
- Example: VirtualPC



31

THE UNIVERSITY OF TENNESSEE KNOXVILLE

In some situations, it's useful to virtualize a guest system that does not share the same ISA with the host system. In these cases, both the application and operating system code for the guest require emulation, for example, by using binary translation.

These systems are called whole system VM's because all of the guest software has to be virtualized. They are typically implemented by placing the VMM and the guest software on top of a conventional OS running on the hardware.

One example of a whole system VM was the VirtualPC software for the older Macintosh machines. This software could run Windows applications on a Macintosh system alongside the regular Mac applications. Its basic design is shown in the figure on this slide. It's no longer used because Macintosh no longer uses the PowerPC architecture.

Co-designed Virtual Machines

- Designed to enable innovative ISA's and/or hardware implementations
- As if the VM software is part of the HW
  - Applications / OS never directly execute native ISA instructions
- Useful for backwards compatibility
  - Example: Transmeta Crusoe

32

For all the VM's we've considered so far, the goal has been functionality or portability – to support multiple OS's on the same platform or to support different ISA's and OS's on the same platform

Co-designed VM's have a different objective. They are designed to enable innovative ISA's and hardware implementations to improve performance or power efficiency.

For a co-designed VM, it is like the VM software is part of the system's hardware design. The native ISA is never actually used by the applications and operating system – they only use the interface exposed by the upper-level VM.

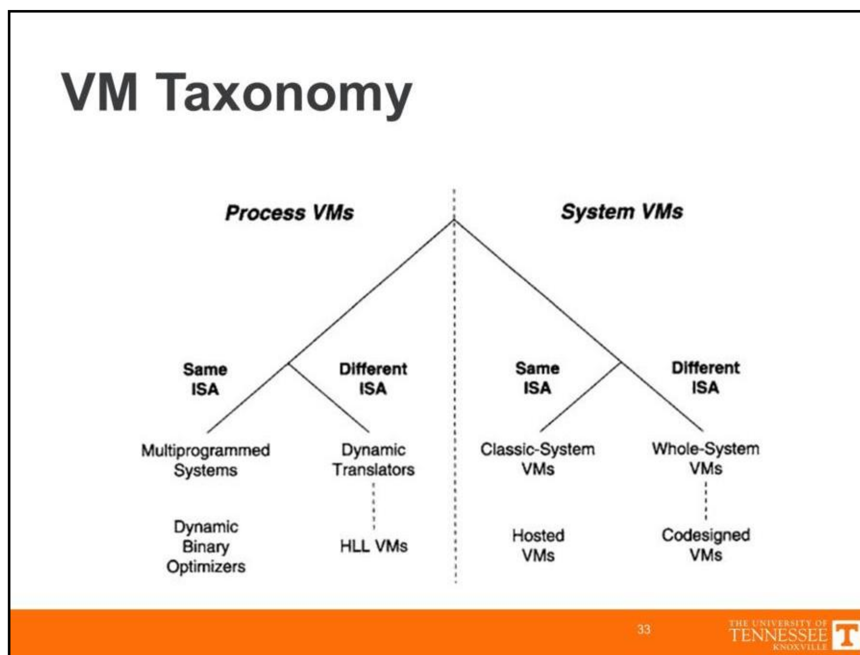
So, think about why this would be useful. I had mentioned earlier that there is relatively little innovation in ISA and operating system design because these things become entrenched and are hard to change. This is mainly because legacy software needs to be compatible with newer systems.

So, say you develop some innovative ISA that is, for example, really power efficient. How could you use it and still maintain backwards compatibility? The answer is to use

a co-designed VM. This is basically what the Transmeta Crusoe system tried.

The Transmeta Crusoe was an innovative processor that used a VLIW instruction set to achieve better power efficiency than Intel and AMD, which used x86-based ISA's. To run applications compiled to Intel's instruction set, they used a co-designed VM on top of the Crusoe. They were able to achieve performance on par with the Intel processors while also reducing power consumption. However, not long after Transmeta entered the market, Intel was able to design its own low-power solutions into their processors, essentially killing Transmeta.

The company still exists – but it mostly just makes money off of its IP.



This figure summarizes the different types of VM we just discussed.

VMs are divided into 2 major types: process and system.
Process VMs virtualize an ABI (user instructions + system calls)
System VMs virtualize a complete ISA (user + system instructions).

And we can further subdivide these types by whether the guest and host use the same ISA or different ISA's.

There are two types of process VM's that use the same sets of instructions for guest and host. The first is multiprogrammed systems, as already supported on most of today's systems. The second is same-ISA dynamic binary optimizers, which transform guest instructions only by optimizing them and then execute them natively.

For process VMs where the guest and host ISAs are different, we also give two examples. These are dynamic translators and HLL VMs. HLL VMs are connected to the VM taxonomy via a “dotted line” because their process-level interface is at a different, higher level than the other process VMs.

On the right-hand side of the figure are system VMs. If the guest and host use the same ISA, examples include “classic” system VMs and hosted VMs. In these VMs, the objective is providing replicated, isolated system environments. The primary difference between classic and hosted VMs is the VMM implementation rather than the function provided to the user.

Examples of system VMs where the guest and host ISAs are different include whole-system VMs and codesigned VMs. With whole-system VMs, performance is often of secondary importance compared to accurate functionality, while with codesigned VMs, performance (and power efficiency) are often major goals. In the figure, codesigned VMs are connected using dotted lines because their interface is typically at a lower level than other system VMs.