

<http://denali.cs.washington.edu/rework/papers/plex86.txt>

Copyright (c) 2000 Kevin P. Lawton Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts.

=====

Running multiple operating systems concurrently on an IA32 PC using virtualization techniques, by Kevin Lawton

(Last updated Nov 29, 1999)

The objective of this text is to bring together a number of ideas on the implementation of virtualization on the IA32 x86 PC. This is a collection of ideas from many people who have offered them to the FreeMWare discussion forum, including myself.

Many folks have contributed to this project thus far. The following is an incomplete list of people who have contributed ideas about the implementation of virtualization on IA32. Email me if I forgot you.

Kevin Lawton, Ulrich Weigand, Ramon van Handel, Cedric Adjih, Clem Dickey, Nick Behnken, Jens Nerche

THE RATIONALE FOR VIRTUALIZATION =====

What many users and developers desire, is a way to run a primary PC operating system and related software, while retaining the ability to concurrently run software engineered for a different PC operating system. For example, one might want to run Linux/x86 as a primary operating system, yet have the ability to run all of their MS Windows applications without the need for a reboot.

There are several strategies that can be used to this end. It's worth describing them briefly, so it's more obvious where virtualization fits in.

Strategy #1, pure emulation:

If you want a solution which runs x86 operating systems and applications on non-x86 platforms, you need to model a fairly complete x86 PC in software, since the x86 instruction set is not available to you. This is the method used by my emulation project "bochs" (<http://www.bochs.com>). The benefit here is portability. The tradeoff is a significant performance hit. Though, there are some advanced techniques, such as dynamic translation which can improve performance.

Strategy #2, OS/API emulation:

Since applications generally run in a different space than the operating system, and communicate via a set of APIs, another method of running applications designed for a different x86 OS, would be to intercept and emulate the behavior of these APIs using facilities in the existing OS. This is the strategy used in the Wine project (<http://www.winehq.com>), though it's complicated by Windows internals issues. Application binaries can be run natively using this strategy, thus one of the benefits is very good performance. Since the OS APIs are emulated, the associated OS is not needed, giving a pleasant side effect of not needing to purchase a license for the OS. On the negative side, this strategy only works for running the x86 OS for which you have implemented emulation of the APIs.

Strategy #3, virtualization:

Let's first talk about why we can't just run two operating systems on the same PC. First, devices such as video cards, disk controllers, timers, etc. are not designed to be driven by multiple operating systems. In general, hardware is designed to be driven exclusively by one device driver. Additionally, System features of the IA32 CPU are designed to be configured and used in a coordinated effort by only one operating system. These would include the paging unit, protection mechanisms, segmentation model, etc. Other application oriented features and instructions are not a problem and

could potentially be used/executed natively. Fortunately these constitute a great deal of the workload imposed on the CPU.

We might generalize by saying that some larger fraction of the feature/instruction set would work in the context of running multiple operating systems, and some smaller fraction would not be compatible. Thus our general strategy for virtualization is to let the larger set of compatible instructions "pass through" and execute natively. We need to detect the set of incompatible ones, and emulate their expected behavior. Thus, this is a quasi-emulation technique.

The x86 CPU architecture doesn't give us the ability to naturally detect usage of 100% of the features which we need to virtualize, so we employ some software techniques to fill in the gaps.

As we certainly can't allow both operating systems to drive the same peripheral IO devices, we require software emulation of a reasonable set of such devices to be driven by the virtualized operating system. Fortunately, years of work on the project "bochs" (a Strategy #1 emulator), have yielded a complete set of such devices, which are known to be compatible with many x86 operating systems. These can actually be shared between both projects, and extended where necessary.

The benefits to virtualization include fairly native performance, the ability to run various x86 operating systems, and no need to depend on the publication of API standards or to keep up with them as they are extended.

POSSIBLE VIRTUALIZATION SCHEMES =====

The ideas and techniques of virtualizing a machine can be carried over to many different architectures, provided they have a capable feature set. Virtualization can also use various schemes depending on the architecture, the operating systems to be run, and other considerations. Some CPU architectures provide virtualization naturally. Others like IA32 require additional software to completely virtualize the CPU.

It is conceivable, that multiple OS environments could be engineered such that they communicated their low-level requirements via a common micro OS which controlled all the devices and system oriented CPU features. If all application and operating system code was well behaved, there would be no need for any additional virtualization, as it would be effected naturally.

Given our real world circumstances, we have to account for OS and application code, which access system oriented features. Our micro OS, in this case, would have to detect and emulate these accesses. Since the hardware devices are among the features which have to be emulated, a like set of devices would be offered to each OS running in this environment, though they don't have to relate to the actual set of devices which are physically part of your particular PC. The micro OS would drive those.

The downside here, is the effort involved in writing such a micro OS, and device drivers for the incredible number of devices out there. Fortunately, we can just use services from a primary OS we run, to implement emulation of the devices which the virtualization needs to offer to any secondary OS we run. This removes us from having to write device drivers for any of the native hardware, and allows us to focus on the virtualization logic.

Given this choice, it's useful to associate some terminology with each role an OS may play. The OS which controls the actual hardware and provides services used by the virtualization code will be called the "host OS". This OS does not need any of the virtualization environment to boot or run. Each OS which runs inside a virtualized environment (emulated devices and virtualized CPU usage) will be called a "guest OS". There can only be one host OS, but conceivably many instances of guest OSs.

CHALLENGE ON THE IA32 =====

A processor could be engineered to be naturally virtualizable. By this I mean, that any instructions which access system oriented features should naturally trap out, giving a virtualization monitor the chance to emulate the expected behavior. It is important to protect reads and writes of system registers, so that poorly behaved application code will also function properly.

Unfortunately, the IA32 architecture is not 100% naturally virtualizable. There are a number of instructions for which write access to system registers is not allowed from user code (this is good; trap generated), but read access is allowed (not good; no trap).

So we must coerce the processor into trapping out when potentially problematic instructions are executed; ones that the processor does not offer us a natural hardware protection against.

In a nutshell, this problem boils down to how to breakpoint on the execution of arbitrary instructions, since the IA32 CPU won't always do this for us. And do this, without the guest OS detecting any changes; otherwise it's execution path could be altered.

If you think about the last statement, it may become obvious that our virtualization objective for the CPU component anyways, is the same as that for a non-intrusive software debugger. How can I set any number of breakpoints on arbitrary instructions, without changing code execution?

With this thought in mind, hopefully the rest should fall into place.

IA32 FEATURES WHICH ARE NOT NATURALLY VIRTUALIZABLE =====

I've put together a list of x86 instructions which need special consideration with respect to virtualization. This should help out in discussion of implementation.

For now, let's assume we're going to use the following strategy. Guest code from all ring levels 0..3 will actually be run at ring3 (the user or application level) so that most sensitive instructions will yield a natural protection exception, and thus we'll get a chance to virtualize it's execution.

Following is the list, as mentioned. A 'Y' in the 2nd column, means that when run at ring3, the IA32 processor naturally generates a protection exception, thus giving our virtual monitor code the chance to do something smart in the context of this virtualization environment. A 'N' means we can't make the processor give us the chance, so we need to figure out how to handle this otherwise. A '*' denotes some special commentary is necessary, and follows after the list.

TABLE 1 -----

TABLE 1

protected in ring3

clts	Y
hlt	Y
in	* (IOPL and TSS permission map)
ins	* (IOPL and TSS permission map)
out	* (IOPL and TSS permission map)
outs	* (IOPL and TSS permission map)
lgdt	Y
lidt	Y
lldt	Y
lmsw	Y
ltr	Y
mov r32, CRx	Y
mov CRx, r32	Y
mov r32, DRx	Y
mov DRx, r32	Y
mov r32, TRx	Y
mov TRx, r32	Y
popf	*
pushf	*

cli	Y (IOPL)
sti	Y (IOPL)
sgdt	N
sidt	N
sldt	N
smsw	N
str	N
verr	N
verw	N
lar	N
lsl	N
lds/les/lfs/lgs/lss	*
mov r/m, Sreg	*
mov Sreg, r/m	*
push Sreg	*
pop Sreg	*
sysenter	*
sysexit	Y

Also of relevance to this topic, are instructions which effect a control transfer or interrupt. The importance of these instructions will become more evident later, so for now I'll just list them:

TABLE 2

call	*
ret	*
enter	*
leave	*
jcc	*
jmp	*
int/into/bound	*
iret	*
loop	*
loope/z	*
loopne/nz	*
wait	*

We can handle the instructions which have native protection ('Y') while running at user-level, by an exception handler which emulates the instruction in our virtualization context. This is exactly what a v8086 mode monitor does. Fortunately, emulation of nearly all x86 instructions is done in bochs already, so there's not much ground to break here.

That leaves the set of instructions with a 'N' or '*' in column 2. Following, I outline special considerations about each of these instructions.

LAR, LSL, VERR, VERW

The worry here is the CPL we are running at, is factored into the behavior of the checks made by these instructions, as are the fields in the segment descriptors, which we may modify according to our virtualization strategies. We don't want the guest to be able to see any modifications we make.

Under the right circumstances of CPL and the descriptors in the current descriptor table, we could conceivably let these instructions execute as-is. Otherwise they will have to be virtualized.

SGDT, SIDT, SLDT

The IA32 architecture let's you store but not load values of certain system registers. In this case, the global, interrupt, and local descriptor tables. This is of concern to our virtualization strategy only if our implementation uses values for these registers, other than what is expected.

SMSW

Moves to/from control registers cause exceptions in user-level code. Unfortunately, SMSW can be run at any privilege level, and reads the bottom 16 bits of CR0. This bits contain the following fields:

- 0:PE protection enable
- 1:MP monitor coprocessor
- 2:EM emulate coprocessor
- 3:TS task switched
- 4:ET extension type (hardcoded to 1 on most processors)
- 5:NE numeric error

Note, I believe bit4 is hardcoded to 1 on Intel processors, though this is not true for some clones.

I think the trick here is that if the virtualization monitor and environment can keep these fields consistent with what the guest OS expects, then we can let this instruction execute as-is.

STR

This stores the segment selector from the task register into a variable. This is of concern to our virtualization strategy only if our implementation uses values for these registers, other than what is expected.

POPF, PUSHF

The arithmetic flags such as CF, should not be a problem, as we're letting user instructions which manipulate them execute natively. Any exceptions generated to emulated system oriented instructions will have to save restore them properly, but that's to be expected. That leaves us with:

- 8:TF
- 9:IF
- 12-13:IOPL
- 14:NT
- 16:RF
- 17:VM
- 18:AC
- 19:VIF
- 20:VIP
- 21:ID

Notes: - PUSHF on the Pentium+ does not copy the values of VM and RF (bits 16 and 17). Instead, these fields are cleared in the flags image on the stack. So in a sense, you don't have to worry about code looking at these fields via a PUSHF for >= Pentium.

```
MOV r/m, Sreg
MOV Sreg, r/m
PUSH Sreg
POP Sreg
```

These instructions load and store segment registers. A concern is the RPL field of the selector. Since instructions which read the value of the selector are not protected against in user code, we need to do one of two things. Whenever possible, we can make sure segments are loaded with a selector which has the RPL which the guest code expects (3 for user code). Or we can virtualize instructions which read the segment selectors, by emulating them and substituting the desired fields. The strategy we

use at any one time, depends on the privilege level of the guest code that is executing as well as other factors.

IN, INS, OUT, OUTS

These instructions have the following protection check (for protected mode), to determine if the instruction will be allowed to make the IO transaction.

```
if ((CPL > EFLAGS.IOPL) && (any corresponding bits in TSS are 1))
    accessible = no
else
    accessible = yes
```

We generally want to reflect IO accesses to our device emulation code. (See section on that later) Thus, we need to insure we always receive an exception when the guest attempts an IO instruction. If the IOPL value in EFLAGS we chose to use at any one time is < CPL (more privileged), then the processor will generate an exception. If we chose to use a value of IOPL such that this is not that case, we can use bits in the TSS to force an exception. We might make a choice to allow an IOPL like this, if the guest is requesting it, and we don't want to virtualize operations which access the EFLAGS register, like PUSHF.

Modifying bits in the TSS IO map, means that we must also virtualize the TSS. (See section on that later)

SYSENTER

This instruction has the following checks:

```
IF CR0.PE == 0 THEN #GP(0)
IF SYSENTER_CS_MSR == 0 THEN #GP(0)
```

So I would say we should do the following. Upon startup of our VM, check what processor we're running on and see if sysexit/sysexit are supported via CPUID.

If not supported, then no big deal. If supported then whenever we warp into our VM, save the value of SYSENTER_CS_MSR and then set it to 0. Then we'll receive a fault when the guest tries to use it. We have to restore the value when warping back to the host OS.

DYNAMIC SCAN-BEFORE-EXECUTE TECHNIQUE =====

As mentioned, one of the key tasks we have, is to protect against the execution of that small set of instructions which do not invoke native IA32 protection mechanisms. So we do it in software.

The path of execution of code can be thought of simply as starting at a well defined address (in the ROM BIOS on the PC), and passing through many branches (jumps, calls, interrupts, etc) along the way. Since we know where execution begins, we can fetch and decode a sequence of instructions up to a branch instruction, and place a breakpoint there. We could then execute the code, which will generate a breakpoint exception at the branch instruction. Our virtualization monitor would receive the exception, effect the branch in the guest code, and given the new target address after the branch, repeat the same process for the next code sequence.

Using this method, we are free to place breakpoints on arbitrary instructions, not just branch instructions. So we can force the guest code to generate exceptions on any instructions, including the ones in the set we talked about.

Essentially, what we are trying to accomplish here, is to never let the execution of guest code pass through unscanned code, otherwise it might run an instruction from that set.

There are some considerations we need to keep in mind, when implementing this technique. First, is that if we use software breakpoints, that means we are physically modifying the code. Conceivably, guest code could read from itself and thus retrieve the breakpoint instruction rather than the original instruction. I'll call this Self Examining Code (SEC). Likewise, we also need the ability to handle when

guest code modifies itself, known as Self Modifying Code (SMC). You could imagine that if we placed a breakpoint at the end of a sequence of instructions, and the instructions in the middle modified the code, we could conceivably lose control of the execution path. So we will detect writes to code which has been scanned, so we can detect this.

And, as discussed so far, the strategy entails continuous decoding, lot's of exceptions and processing, and thus very non-optimal performance. So as an extension, wherever possible, after we've scanned code sequences, we attempt to let them run without intervention on the next pass. Ideally, we want to only scan code which exists on code paths which haven't been taken before.

The paging mechanisms are a convenient way to protect against reads/writes to small regions of memory on IA32, something we need to do to handle the considerations above. So we make use of paging protections.

Let's start by thinking about some code contained completely within a linear page (and it's related physical page). Let's say execution begins at instruction i0, somewhere in that page. The first revision of our strategy might be as follows. We start parsing instructions, beginning at i0 until we encounter:

- - An instruction which is in our current list of those which can not be run natively
- - A branch instruction
- - The address of an instruction sequence which has already been analyzed

For the first 2 conditions we install a breakpoint at the beginning of the terminal instruction. For the last condition, we don't need to do anything, since the following code has been dealt with already, breakpoints installed downstream wherever necessary.

We then allow the code to execute natively. Execution continues until it reaches a breakpoint condition we set. If we have encountered an instruction which can not be run natively, then we emulate it's behaviour and begin analyzing the instructions which proceed it, as above. If we have encountered a branch instruction, then we could single step through it's execution and begin analyzing as above starting with the target address. Additionally, if the target address is non computed and has been analyzed and marked OK already, then we could mark this branch instruction as OK, and let it execute natively from now on. Computed branch instructions, we need to monitor. Since the target address is dynamic, we don't know if it will branch to a sequence we have scanned before.

Thus, we are dynamically monitoring code, making sure we never let execution branch to code we have not yet examined.

Our strategy so far, does not address the possibility that some instructions which write to memory, may write into our page of code, specifically into the address of instructions which we OK'd and allowed to execute natively. This is where the paging system comes in handy. Using the page tables, we can write protect any page of memory in which we have analyzed and OK'd code. Since multiple linear addresses can be mapped to a single physical page, for perfection, all page entries pointing to the physical page with trusted code would have to be write protected.

Then, upon a write-protect page fault, we have the opportunity to unprotect the page, step through the instruction, do something intelligent with respect to the meta information we store about that code page, and re-protect the page. We might for example, just dump the meta information about that code page, and start from scratch.

I outline the steps required by such a technique in more detail below, as well as some possible implementation details.

- For each new code page we encounter, allocate a page which represents attributes of the instructions within the page. Zero out page to begin with.

- Each byte in this corresponding attribute page, denotes attributes of the instruction which starts at that offset in the code page. Here's a possible layout of the bitfields in each byte:

```

7 6 5 4 3 2 1 0
|||||
||| | +-+ +-+ +---- instruction length 1..15
||| |
| | +-+----- available for future use
| |
| +------ 0=execute native, 1=virtualize
|
+----- 0=not yet scanned, 1=scanned

```

When bit7 is 0, all the other bits are meaningless, since we have not yet scanned the instruction.

- At first, when we encounter new local-page branch instructions (static offsets) the target address in the page may very well not have been scanned yet. We could mark this instruction as one to virtualize for now. The virtualization logic could simulate the branch until the target address has been scanned, in which case we could mark the instruction to execute natively from thereon. Lazy processing at it's best.

The next step beyond this strategy would be to use a recursive descent technique and branch out pre-scanning the one (unconditional branch) or two (conditional branch) possible target addresses. Upon returning from the recursion, granted both are in the local code page, we would likely be able to let the branch instruction execute natively. Code downstream could generate breakpoints where necessary. Terminals in the recursion could be:

- instructions which are already scanned
- out of page branches
- instructions which require virtualization, though we could scan right through these.
- instructions whose opcodes cross page boundaries.

We may also want to establish a maximum recursion depth. The win we get is if the code we prescan is well used before we have to dump the attribute page. We lose when this is not the case, for instance when the code page is modified frequently via self modifying code or due to code pages which share data. We may find a comfortable max depth of N, which does a much better job winning than losing on the average, through trial and error.

- When we detect a write to a code page for which we have scanned code, there are multiple actions we could take.

We could simply dump all info for that page.

Or we could conceivably examine the address and data size of the instruction's access in the page-fault handler, to determine if it stepped on addresses for which we have scanned instructions. Upon examination of the affected addresses, we'd look at the corresponding attributes. If they pertain to an instruction(s) which we have pre-scanned, then we need to dump all the mappings for the entire attribute page. This is because the technique I have here doesn't record which instructions branch into these addresses, so there is no way to know which other instructions to invalidate. This is the tradeoff for a simple algorithm. If such writes go to areas in the page which are not yet marked as pre-scanned, then we can step through the instructions. We can consider the writes as ones to data in a shared code/data page. And as such, we don't need to dump the page attributes we have accumulated thus far.

For simplicity, let's start with the first option.

- We need to cope with out-of-page branches and computed branches, in a way such that we never lose control of execution.

The simplest approach is to mark such instructions as needing to be virtualized the monitor. We should chose this one as our first approach, for simplicity.

For static offset out-of-page branches, if we wanted to go a step further and let them pass-through, we would have to keep some additional page/branch information, which can add an amount of complexity to our monitor. The main issue is that we may have to invalidate code in a given page that is branched to. If we were to allow long static branches to this code, then we would have to iterate (or recurse) through all the code pages which had such branches, and do some invalidating there. This would require a very efficient "edge list" between pages etc.

- This technique handles overlapping instructions well. Each byte in the attribute page holds info about only the instruction which *starts* there. So there can be attributes for instructions which start in the middle of a previous instruction with no conflict. This technique also makes things simple for the code which handles writes to a code page. Given the affected addresses, you can easily scan forward and backward to see if a pre-scanned instruction was hit, and then invalidate the attribute page. Or in other words, it can handle self-modifying code well.

- This technique extends out to code which spans multiple pages, quite well. We have an additional boundary case where an instruction crosses a page boundary. In this case, we need to write-protect both pages, and handle modified code across both pages. And the tables we use to keep track of which instructions have been analyzed, will likely be oriented (hashed) in a way that factors in page size for performance reasons.

Or if we want to take the easy way out, we could just mark any instruction which spans 2 pages, as needing virtualization. We could then just step through the instruction, and resume scan/executing thereafter.

- This technique eats up memory as we encounter and monitor new pages. So we'd have to have some threshold at which we could dump old private pages, and the associated info about which instructions in them have been monitored, so they could be reused. Probably a good candidate for an LRU strategy.

PROTECTING AGAINST GUEST READS TO CODE THAT WE MODIFY

=====

We previously talked about how to monitor code before it is executed, in order to virtualize arbitrary instructions (usually ones that don't offer natural protection). With this technique, we make use of breakpointing.

Hardware breakpoints would seem like a natural choice, since they are non-intrusive. However, there are a limited number of them (only 4), and their use by the virtualization code potentially competes with use of hardware breakpoints in the guest OS. I'd like to offer the ability to allow guest OS hardware breakpoints. So we need to be prepared to use software breakpoints, if/when any of these factors make it necessary.

Software breakpoints (INT 3, single-byte opcode 0xCC) give us the ability to install unlimited breakpoints in the code we monitor. However, a side effect is that we have to modify the code by inserting them. This offers a potential for incorrect execution when running any code which depends on a read of executable code being completely accurate.

Access through any of the data segment registers could potentially read from a section of code we have modified, and "see" the software breakpoint instructions we have installed. Unfortunately, the paging protection does not offer a natural differentiation between reading and executing code. If it did, we could use it to protect against reading from a page which we have modified, while at the same time allowing execution in that modified page. Then upon a read, we'd receive a page fault, and could spoon feed the read data, according to the unmodified page. Thus it would never see the modifications.

The next section, explains a way to exploit the separate I&D TLB caches, to do something similar to this. Separate caches did not exist on earlier processors like the 386 and 486. So here's an alternate strategy.

What we can do is to execute code from a private copy where the modifications are made. Reads would occur from the actual code; execution from a private modified copy.

In the virtualization technique we talked about previously, we write protect pages of code which we are monitoring. Since we are notified of changes to the page via a page fault, we have the opportunity to propagate the change to the private copy as well, and take appropriate actions.

What we could do, is to provide a separate code segment (CS) descriptor, for the purposes of pointing into our private modified code page. When our monitor effects control transfers back to gusted code, this descriptor will be fetched and loaded into CS. Other segment registers, like DS, will be loaded from descriptors which point to the normal guest data space, including possibly the original code page.

A concern we would then have is that instructions which access memory, may override the default DS descriptor access and use CS (override opcode 0x2E). We don't want any reads accessing memory via our private code descriptor. This can be addressed by making sure our modified CS descriptor is marked as execute-only. Attempts to use CS for reads will then generate exceptions. Or we can virtualize all instructions which use a CS prefix opcode.

(See Caveat #1)

USING THE TLB CHARACTERISTICS TO PROTECT AGAINST CODE READS/WRITES

=====
If we could let code execute in a page, but disallow reads/writes to the same page, we would be notified when have encountered self-examining or self-modifying code.

As it turns out, most new IA32 CPUs from Intel and other vendors, have split I&D TLB caches for better performance. That is to say, they cache values from page tables in a separate instruction TLB cache as instructions are encountered, than from similar values in a data TLB cache as they are encountered.

It is possible under the right circumstances, to use this split cache to our advantage, so we can effectively have the ability to run in code which can not be read or written, when it is available on the current CPU. This is generally available on anything Pentium and beyond and clones of such processor levels, as a test program which was run by many users confirmed. This technique is known `_not_` to work on the following chips, likely due to a combined I&D TLB cache. We could even dynamically determine if this technique is available to the monitor, given the CPU it is running on.

TABLE

386 (doesn't even have INVLPG)
486-DX66
Cyrix PR150+
Cyrix PR200+
Cyrix MediaGX233
Cyrix M2 PR300 MX
Cyrix M2 333
AMD K5 PR100
IDT C6 200
NexGen Nx586 100

The technique works as follows (from the monitor's perspective).

- invalidate the code page with INVLPG. (Wipes I&D TLB entry clean)
- make sure page table entry is ring3 accessible
- create a private mapping to the code page (different linear address maps to same physical)
- write a RET instruction into the code page (using private mapping)
- call the RET instruction using normal mapping (Loads I TLB entry)
- replace the instruction where the RET went, using private mapping
- set the page table entry permissions to only ring0 accessible.

- switch to the ring3 code.

Essentially, we have loaded the instruction TLB cache entry for a particular code page, with a ring3 accessible TLB entry. But before we transfer to the ring3 code, we have modified the entry to be not-accessible anymore. Since we originally flushed the TLB entry, the data TLB cache entry is invalid. In the future, a data access will attempt to load the page table entry from the modified entry, which is not ring3 accessible. Thus a data access, be it read or write, will generate a page fault - our notification and a chance to do something about the fact that the guest code is reading/writing in the same page. Yet the code is able to execute, fetching using the page translations and permissions cached in the instruction TLB cache.

Note also, that there is no deterministic guarantee how long the TLB entry will be maintained for. The CPU is free to dump it at any time. But this doesn't provide a problem in our strategy, other than performance. Next time the TLB entry is loaded, it will read in the entry which is non-accessible from ring3 and provide an exception. At that point, we can re-iterate this process.

LDT/GDT/IDT VIRTUALIZATION =====

Let's assume that the host OS has its own complete set of local, global, and interrupt descriptor tables, as well as page tables. For our virtualization techniques, we need flexibility to maintain a separate set of these tables. We need to play tricks with the page tables, and to create descriptor tables, such that we can at times let guest code load descriptors using the natural x86 mechanisms. We also need to create our own interrupt descriptor table, to handle exceptions generated by the guest code as part of our virtualization, or just as part of natural exceptions/interrupts which are generated by guest code.

To have such flexibility, we need to save the host context of these tables, and switch to a completely separate context whenever we run our guest code for a timeslice. Then restore back to the host context when our timeslice is done. In a sense, our monitor code is running the guest OS/code, in a "parallel" context to the host OS.

[I plan to fill in more here later +++]

VIRTUALIZING DESCRIPTOR LOADING =====

(for protected mode guest code)

Granted we virtualize all protection levels of code (0..3) by running them at ring3, we will have a privilege level mismatch with respect to loading of segment registers when not running ring3 code.

While running user code (ring3), the descriptor loads should execute as expected, due to the match in effective and actual privilege levels. This is provided, we point the GDTR and LDTR registers at descriptor tables which contain descriptors that the guest code expects.

While running guest system code (rings0..2), an exception would be generated whenever loading from a descriptor that has a privilege level < 3, whereas the load may have normally succeeded.

One way to solve this, is to protect against instructions which load the segment registers, when running code effectively at CPL of {0,1,2}. Then emulate them. We have to virtualize instructions which examine the segment registers at these privilege levels anyways (because they may look at the RPL field which will not reflect the expected privilege level), so doing the same for instructions which load them is just an extension.

What I'd like to explore further is the idea of using a private GDT and LDT for the virtualization of code at CPL {0,1,2}. If we virtualized instructions which looked at the GDTR and LDTR, we could load them with values pointing to private copies. The private descriptor tables could start out empty, generating exceptions upon segment register loads. Each time, the exception handler could generate a private descriptor which would allow for the next segment register load to execute natively. For example, say we are running guest code which is effectively at ring0, but really at ring3. And a segment register load attempt occurs, for a ring0 segment. The first attempt, would invoke an exception, where we could build a private descriptor which actually had a descriptor privilege level of ring3. The next attempt would work natively due to the protection level match of our private descriptor.

Each time the descriptor table registers (GDTR and LDTR) are reloaded, we could dump our private descriptor tables and start anew. Our choices are to start from empty tables, or map all the real descriptors to private ones in one shot.

In order to implement such a technique, we would need to make use of the paging unit protection mechanisms. We need to be notified when system code changes a descriptor table entry, so we could re-adjust (or dump) our private copy. We need to page protect any pages which contain the GDT and LDT descriptors. The exception handler would recognize that a write access to descriptor table memory occurred, and do the correct thing with our private descriptor tables. We have to keep the accessed bit in each of the descriptor tables correct, so it may be better to start out our private descriptor tables with empty descriptors, and change the accessed bit in the real tables during the exception handler where we build a private entry. If we build them all at one time, then the accessed bit would only be modified in the private copies, upon future loads.

OVERVIEW OF VIRTUALIZATION =====

Please refer to the following table. It should help visualize where the various components of our virtualization strategy lie.

TABLE

HOST OS CONTEXT	GUEST OS CONTEXT
Ring0: host kernel/monitor module	monitor kernel
Ring1:	
Ring2:	
Ring3: monitor app/IO emulation	guest OS kernel + app

In this table, each "context" consists of both kernel and application code, it's own set of page tables, descriptor tables, and other system mappings.

We run our monitor application in the host OS, like any other program. One of it's chores is to maintain the emulation of IO devices. Since we are emulating the devices at the host OS user level, we can make full use of all the host OS facilities that are available to any user program, for instance libc calls, GUI calls, etc. The host OS will natively schedule this monitor application task on/off the processor for time quanta, using it's scheduling algorithm.

The second chore, when not handling IO emulation, is to make sure the guest OS gets some time on the processor. To do this, the host OS monitor application requests that the guest context be run for a time quantum, via a system call which is received by the host OS monitor module.

Since this kernel module is privileged, it can store the necessary current host OS context, and switch over to the guest OS context.

Over in this context, let's assume we have "pushed" guest OS kernel code from ring0 to ring3, to assist in virtualization. Other discussion explain why we do this. The context switch leaves us in the monitor kernel, which can then run the guest code.

Within the guest OS context, there will be potential exceptions generated due to virtualization of certain features. These will be handled internally by the guest OS monitor kernel. How the guest OS monitor kernel handles things can be controlled by calls from the host OS monitor application via the host OS monitor module, which can tweak things over in the guest OS context.

It is important to understand that the guest OS context does not handle managing real hardware, yet real hardware interrupts may occur during execution within the guest OS context. We must reflect these interrupts to the host OS, so it can handle them promptly.

When the guest OS monitor kernel receives an interrupt that was meant for the host OS (timer interrupt for instance), it switches back over to the host OS context and let's it handle the interrupt.

The issue here, is that the next time we get a time slice from the host OS scheduler, we need to make sure it switches us to the context of the host OS monitor application, since execution of the guest OS kernel and application code (though at ring3) is not executing within the proper context. So in the guest OS monitor kernel, we need to fake a return from the `system_call()` listed above such that we'd be back in the host OS monitor code and host OS context, before switching back to the host OS to handle the interrupt.

In the next time slice our host OS monitor app gets, it will again make the `system_call()`, and we'll do it all over again, maintaining proper contexts without having to modify the scheduling algorithms in each host OS kernel.

SOME NOTES ABOUT HOST AND GUEST PAGING

=====

So far we've hammered out much of the framework for virtualization, but haven't talked about an important topic, paging. Each of the OSs (host or guest) would have both a physical and virtual memory footprint if it was run on a real unvirtualized PC. You probably have your computer configured with enough physical memory as to have reasonable performance. Your OS chews up some memory for locking down the kernel, and multiplexes the rest for virtual memory.

Now comes along a second OS. Let's say we didn't have any kind of paging on the whole amount of memory needed by the guest OS context. We'd need to acquire this memory via the host OS kernel, so it doesn't use it for other purposes. It would have to all be locked down. This would tremendously decrease the amount of physical memory resources available for the host OS environment, and thus would be incredibly detrimental to it's performance.

One option would be to lower the amount of physical memory taken by the guest OS context and let the paging system in the guest OS handle virtual memory. This of course makes the performance of the guest OS worse, the cherry on top being the extra overhead incurred due to the framework in the guest OS monitor kernel. So we walk a tight rope here.

So I'm throwing out the following idea. If we could efficiently coordinate parts of the page tables between the host OS context and the guest OS context (remember we're using different page tables too), then perhaps we could run the guest OS monitor kernel in locked down memory, and the guest OS kernel + app code (which is being run at ring3) in pageable memory.

When a page fault is received by the guest OS monitor kernel, we could detect that it is due to a real page not present condition, and reflect that exception back to the host OS kernel, the same way we would when we reflect an external interrupt condition. What we are achieving here is a smaller physical memory consumption on the host OS, and use of the host OS's native paging system, so that the guest OS only hogs memory resources when it uses them. When it's idling, a large part of it could be paged out.

I'd like to get people's reaction on this stuff. Let's iron out the wrinkles in this, and then I'm afraid, we're going to have to start writing some code... :^)

MAPPING THE MONITOR'S GDT AND IDT INTO THE GUEST LINEAR SPACE

=====

The monitor is never directly invoked (called) by the guest code, in fact the guest shouldn't even know about the monitor. The monitor is only invoked via interrupts and exceptions which gate via the monitor's IDT. And since entries in the IDT point into the GDT, we need to look at both, with respect to where to map them into the current guest task's linear memory.

As our IDT and GDT must occupy the same linear address domain as the guest code which is normally executing, we need to make sure there are mechanisms to allow these structures to

cohabitate with the currently running guest task's address space. And keep in mind, there can be N different address spaces, depending on which guest task is currently running.

If we virtualize these structures, we need to maintain both the guest's copies of them, and modified working copies of such structures, which are actually used by the processor. When the guest OS accesses these structures, the monitor will receive a page fault, since we need to protect the pages which contain the guest's copy of them. Upon receiving the interrupt, the monitor can update the working copy used by the processor, accordingly.

Another point worth noting is that the SGDT and SIDT instructions are not protected and thus ring3 (user) code may execute them. They each return a base address and limit, the base address being a `_pure_` linear address independent of the code and data segment base addresses. To offer really precise virtualization, in the sense that the user program will not detect us influencing the base linear address at which we store these structures, we could use the 2 following approaches.

Approach #1:

If we are performing the pre-scanning technique, we could simply virtualize the SGDT and SIDT instructions, and emulate them to return the values which the guest code expects. In this case we can place the GDT and IDT structure anywhere in linear memory such that they are in an area which is not currently used by either guest-OS or guest-user code. We have access to the guest page tables, so it is fairly easy to find a free area.

Approach #2:

Under certain circumstances, we may be able to locate the working copies of the GDT and IDT structures, at the linear addresses requested by the guest via loads of the GDTR and IDTR registers. If we can do this, then we may let execution of these instructions pass through without intervention.

This is a condition that is likely to occur while running guest application code. It is common for an OS not to allow access from application code to these structures. Given this is the case, we can map our working copies into the current linear address space, right where they are expected to be.

MAPPING THE ACTUAL MONITOR INTERRUPT HANDLER CODE INTO THE GUEST LINEAR SPACE =====

Now that we've discussed placing the GDT and IDT in linear memory, we need to map the actual interrupt handler code as well. Since we will be virtualizing the IDT and GDT, the guest OS will not see our segment descriptors and selectors, so we have some freedom here. We can place this code (by page mapping it) into an unused linear address range, again given we have access to the guest-OS page tables.

The interrupt handler code, is actually just code linked with our host OS kernel module. The consideration here is that code generated by the compiler is based on offsets from the code and data segments. This code will not be calling functions in the host-OS kernel and should be contained to access within its own code and data when used in the monitor/guest context.

So we must set the monitor's code and data segment base addresses such that the offsets make sense, based on the linear address where we map in the code. For example, let's say our host-OS uses a CS segment base normally of `0xc0000000` (like previous Linux kernels) and our kernel module lives in the range `0xc2000000 .. 0xc200ffff`.

Then let's say that based on empty areas in the guest-OS's page tables, we find a free range living at `0x62000000 .. 0x6200ffff`. We would make the descriptor for our interrupt handler contain a base of `0x60000000`, so that the offsets remain consistent with the kernel module code.

And of course, we mark these pages as supervisor, so that in the case they are accesses by the guest OS, a fault will occur. We will also be virtualizing the guest-OS page tables, protecting that area of memory, so we can update our strategies. Thus, we will know when the guest-OS makes updates to its page tables. This gives us a perfect opportunity to detect when an area of memory is no longer free. If the guest-OS marks a linear address range as not free anymore, and that conflicts with the range we are using for our monitor code, we can simply change the segment descriptor base

addresses for code and data, and remap the handler code to another linear address range which is currently free. No memory transfers occur, only remapping of addresses.

This kind of overhead will only occur once per time that we find we are no longer living in free memory. To reduce this even further, we could start out at, and use alternate addresses, which are known not to be used by particular guest OSes.

VIRTUALIZING THE TSS =====

[I plan to fill in more here later +++]

TIMING ISSUES =====

As we are allowing a great deal of code to run natively, emulation of the system timers (PITs) should be highly accurate as well. Keep in mind that the Programmable Interrupt Timers, like other IO devices need to be emulated use in the guest OS, since they are in use by the host. Let's assume that we're not trying to multiplex the real timers, and that we do need to emulate them.

We could generalize execution of guest code by saying that each time slice of guest code execution is bounded by an exception of some kind. It may be generated by the host OS system timer, telling us our time slice within the host OS context is over. It may be due to a breakpoint or other protection installed by the virtualization, so we have a chance to emulate a behavior. Or due to a true exception occurring within the guest OS.

At any rate, the exception will vector to a routine in our monitor's IDT. We need a mechanism for measuring the time between such exceptions to facilitate an accurate timer emulation. On Pentium+, a very good choice would be to make use of the performance monitoring counters.

The RDTSC (Read Time-Stamp Counter) instruction will give us an accurate reading which we can use. It is also executable from CPL==3, given that CR4.TSD==0, so we could use it efficiently in user-level monitor code if necessary.

The only issue I can think of, is that this would nail us down to Pentium+, as the 386 and 486 don't have these performance counters. I'd like to support these chips if possible, so perhaps we can conditionally sample the real PIT instead. This will give us much less resolution, but perhaps it's a reasonable alternative, their execution speed is correspondingly lower.

Even if the TSC (Time-Stamp Counter) is in use by the host OS, we should be able to multiplex it's use, between the host and monitor/guest environments. If so, we need to save and restore it to appropriate values, as part of our warping to/from the host/monitor contexts.

We will have to build a certain amount of low-level time facilities into the monitor. Our PIT emulation as well as emulation of other IO devices can make use of these high-resolution facilities.

There are two main software components in our virtualization strategy. We have (1) a user program component which communicates to (2) a kernel module component through the normal kernel interfaces like ioctl(), read(), write(), etc.

All or most of the device emulation (video board, hard drive, keyboard, etc) will be done at the user program level, and we'll make use of the standard C library interfaces and such to implement these.

The reason I say most, is that for performance reasons, parts of all of some particular devices such as the timer and interrupt controller chips can likely be moved into the monitor domain. As was talked about before this would alleviate a lot of context switching between the host/guest contexts. We don't have to do this kind of thing right away. Though, it's worth pointing out that parts of quite a few devices can be moved into the monitor. For example the floppy controller could be done in the monitor, the floppy drive in the user program. The VGA adapter in the monitor, the CRT display in the user app. Etc. etc.

Anyways, so we need some kind of accurate time reference and timer services from this virtualization framework. For example, to emulate the CMOS RTC, you need to be notified once per second so you can update the clock. Because of these needs, we need to develop such a framework.

Our timer stuff has to relate very closely to the amount of real execution time the guest code has. What we don't want to use are time references based on the host OS system, as those are highly dependent on system load and other factors. Depending upon the guest code running, there may also be a considerable amount of time spent in the monitor as part of the virtualization implementation, for certain local chunks of guest OS code. We should exclude this time if at all possible, since it is not time when the guest OS is really running, and it would skew the time reference.

So our approach could go something like this. Each time, just before the monitor hands over execution to the guest code, we take a snapshot of time, using the RDTSC instruction. Linux even defines an asm macro for this. :^) Upon the next time invocation of our monitor code (via an interrupt or exception) we take a 2nd sampling using the same instruction. Now we have an accurate time sample of how long the guest code actually ran without intervention. We pass this duration to the timer framework. If there are requests from the device models to be notified given the elapsed time, then we call them. If they live in the user app world, then we return back to the user app, which sees this as a return from the ioctl() call, and some fields are filled in, like how long we ran for etc. If we were wicked perfectionists, we could subtract off the number of cycles it takes to get the guest code started again, and for the exception to occur from our RDTSC values.

Of course, guest code we run at any one time could conceivably not invoke the virtualization monitor, before our next device model requests being notified. The next bounding event would be caused by a hardware interrupt redirect. Each host OS can have set the IRQ0 timer to interrupt at a particular rate, but let's say it's 100Hz or every 0.01 seconds. Let's say a device model wants to be woken at say 0.005 seconds, and that some guest code runs which is not naughty, and doesn't invoke the monitor during the next user process time quantum.

So if we wanted highly accurate timing, we need a mechanism for interrupting us in the middle. Fortunately, the built-in APIC on the Pentium has a timer based on CPU clock speed which can do this. It can be programmed to either periodic or one-shot mode. (thanks to one of the developers for suggesting use of this timer facility)

If we saw this condition, we could set the APIC timer to go off at the equivalence of 0.005 seconds, and our monitor will be notified right on the money. Other tricks like temporarily reprogramming the PIT or the CMOS timer for a finer grained interrupt during that one quantum could be used as a back-up plan for CPUs without the APIC timer capability. For these CPUs, rather than getting a time reference by way of reading the time-stamp-counter with RDTSC, we could read the PIT counter register. This is not as high-resolution, but perhaps functional enough.

I suppose, for starters we could declare the resolution of our timer facilities to be, at best, the interval of the host OS's periodic interrupt rate. :^)

To tie this together with the FreeMWare code we have already, let's look at how this plays out for another contrived example. Again, the host OS uses a 0.01 second periodic interrupt. And let's say the next interrupt required is at 0.035 seconds. The user app code component would probably look something like this:

So far all time reference has been relative to the execution of guest code. This is the accurate way to make things respond to the guest code properly. There are however, things which are better tied to the host OS time reference.

Let's pick on the VGA emulation. There really are two parts to it. The hardware adapter emulation is the first. It needs to live in the time reference of the guest code for accurate emulation. It does not care if there is a CRT attached to it or not, or in other words whether you actually view the output or not. The emulation of spewing the frame buffer output to your CRT can be done in any time reference. This will be implemented by using a GUI library, on a lot of platforms X11, and at the user application level. You might want to refresh the output every so often, if it is updated, but not too often otherwise you'll bog the system down.

In this scenario, we are better off using timing facilities of the host OS. It's probably better to move this function off into a separate thread/process. There are other device models which are candidates for this sort of separation. We can look into this more as we go.

TRANSITIONING FROM THE MONITOR TO THE HOST LINEAR ADDRESS SPACE

Yes, these are some good comments. I've been down this road and found the same. There are some hacky ways I was thinking of, for utilizing tasking, but getting back to host context is worse since you don't have the same room to play.

The simplest way is to have a host<->monitor shim which sits within a single page of memory. The address in the host world is just the linear address where insmod places your module code. When you're in the guest and you want to get back to the host, you make a quick switch of that page mapping to make it the same physical address as in the host, invalidate the TLB entry for it, then jump to it. (remember the differences in CS base values though!)

A similar process happens in reverse. After you've made the transition, you have to restore the page mapping back to the way the guest wants it. During normal operation in the guest, the page of shim code does not exist in the linear addressing world, or at least it doesn't need to.

I call this the "worm hole" technique. That one page (it could be more but you shouldn't need it) is the nexus between the host linear world and the guest linear world. You open it up, make your quantum jump into the next world, then close the worm hole behind you.

VIRTUALIZING THE GUEST OS PAGE TABLES

To implement our virtualization framework, we need to make heavy use of the CPU's native paging system. We use it to protect pages against being accessed by the guest code, and to play other tricks. The guest OS page tables represent the way it expects to allocate, protect, and map linear to physical memory when it is running natively on your machine and at expected privilege levels.

Note that (as per our current strategy), before we even begin execution of the guest environment, our host kernel module allocates all of the physical memory for the guest virtual machine. This memory is non-paged in the host OS to prevent conflicts. As such, we can take some liberties with the paging mechanisms, using paging faults as a mechanism for the monitor to intervene when the guest is accessing data which we need to virtualize.

Since we are virtualizing the guest OS code, changing privilege levels it runs at, changing the physical addresses at which pages reside, and using the paging protection mechanism for other tricks, we can not just use the guest OS page tables as-is. We must use a modified version of them, and be careful that these modifications are not visible by the guest OS.

What better mechanism to virtualize the guest page tables than the paging unit! We can use protection flags in the page table entries to be notified when the guest OS attempts an access to the page tables. At this point, we can feed it the data it expects (read), or maintain the effect that it intended (write). In this way, the guest never senses our modifications.

When there is a change of a page directory or page table entry by the guest OS, we look at the requested physical page address, and can map this to the corresponding memory allocated for the guest VM by the host OS module.

Our page fault handler must look at the address to which an access generated the fault, and determine whether it is a result of our monitor virtualizing system data structures such as the page tables, descriptor tables, etc (in which case the access is valid but we need to feed it fake data), or the access was truly to a non-existent or higher privilege level page (in which case we have to effect a page fault in the guest OS).

One of the behavioral differences that results from us "pushing" ring0 guest OS code down to ring3 to be run within the VM environment, is that we no longer have the dual protection page level access capabilities which the guest OS expects. The paging unit classifies all code in rings{0,1,2} as being supervisor and can essentially access all pages. Code which runs in ring3 is classified as user code and can only access pages at that level. But if we push all rings of guest code to ring3,

```

guest monitor
0 --+ 0
1 | 1
2 | 2
3- -+--->3

```

then there is no longer access level distinction between guest OS code and guest user code.

There are a couple fundamental approaches we can take to solve this behavioral difference.

APPROACH #1

For each set of page tables which is encountered while running the guest OS, we could maintain two virtualized sets of page tables. A first set would represent the guest's page tables biased for running the guest *user* app code within our virtualized environment. Since we'd be running ring3 code at ring3, this would be fairly straight-forward. Only those pages normally allowed access from ring3 would be allowed access from guested user code.

A second set would represent the guest's page tables biased for running the guest *system* code within our virtualized environment. Since the guest's system code expects to be able to access all pages, we can mark system and user pages all as user-level privilege in this set of page tables - except where we protect otherwise to implement various virtualization tricks.

APPROACH #2

Rather than push guest system code down to ring3, we could push it down to ring1 instead. This would yield a privilege level mapping such as the following.

```

guest monitor
0- -+ 0
1 +--->1
2 2
3----->3

```

As far as page protection goes, this would offer us an environment, where we could use the page protection mechanisms more natively, and which would only require one private set of page tables per each guest set. Since x86 paging groups all CPLs of {0,1,2} as supervisor-mode, the shift from 0 to 1 in the example above does not affect privilege levels with respect to paging.

All the instructions, where the ability to execute them is based solely on CPL, generate an exception when not run at ring0. So execution at ring1 will have the same effect for those instructions, as from ring3.

The question is, which other virtualization strategies does this interfere with? Well, an obvious point, is that since we're modifying the RPL of the selector from 0 to 1, we need to virtualize instructions which expose can expose the RPL. So, instructions such as "MOV AX, DS" and "PUSH ES" need to be virtualized.

[I plan to fill in more here later +++]

(See Caveat #2)

Since executing privileged instructions pushed down to ring1 will result in an exception anyways, we could chose to virtualize them instead, and run them at their original privilege level:

```

guest monitor
0----->0
1----->1
2----->2
3----->3

```

Perhaps we would be able to gain something, in the way of allowing certain descriptor accesses to occur naturally. We can certainly only do this, when our virtualization is controlling execution, by way of the scan-and-execute technique. We must never allow execution to reach and execute a privileged level instruction natively. Or other non-privileged yet sensitive ones.

[I plan to fill in more here later +++]

MAINTAINING THE VIRTUALIZED PAGE TABLES

Regardless of the method chosen, we still need to define how the page tables are maintained. Within a hypothetical guest OS, there could be N active page tables, let's say one for each running task. Note that there may be one or more common regions in the linear address spaces described by these page tables. This would be the case, for instance, for situations where each application has its own mappings for a particular linear region, and the OS code maintains a consistent set of mappings in a different linear region.

When the guest OS attempts a switch of page tables (generally associated with a hardware or software task switch), the monitor will intervene and effect that change in light of our virtualization strategies. Again, we have some options to discuss. In short, we can either dump old page mappings and start anew upon each reload of the PDBR, or try to be smart and store multiple sets of page tables.

The simplest approach is to dump the old page table mappings each time the guest requests a reload of the PDBR register. One could imagine that we could then mark all the entries in the virtualized page directory as being not present, so that we could dynamically build the virtualized page tables. Each time a new directory entry was accessed, the page fault would be handled by the monitor, which could in turn mark all the page table entries not present except for the one it needed to build. (we of course except the small region where our interrupt and exception handlers have to exist) Using this technique, we could also dump our page mappings during a privilege level transition. If we chose to do this, then the issues above regarding which privilege level to push guest system code to, are moot since we just rebuild the page tables dynamically according to the effective CPL of the guest code.

A more complex approach would be to save virtualized page table information across PDBR reloads. The idea here is that when the guest OS schedules a task whose page tables are already virtualized and stored, we can save a number of page faults and execution of associated monitor code, which would otherwise be incurred from the dynamic rebuilding of the page tables. This technique does generate some issues. One is that it requires more memory for storage of additional page tables. It also requires additional logic to keep track of the page tables, and must properly maintain situations where parts of multiple page tables are shared.

THE ACCESSED AND DIRTY BITS =====

If we maintain a private copy of page directories and page tables, it is not enough to only monitor changes in the guest's tables and modify a private copy accordingly. The accessed and dirty bits (only the accessed bit in the directory) allow the CPU to give feedback to the OS on the use of page directory and page table entries, by updating these fields. We must therefore insure that we provide coherence between these flags as they are updated in the private tables by the CPU, and those that are in the tables provided by the guest which we are virtualizing. We certainly can't or at least don't want to have to do this on an instruction by instruction basis.

Two good places to make this update are upon the guest's access to the page tables, and at a time when we "dump" virtualized page mappings for a previous set of page tables. To allow the monitor a point of intervention, when the guest accesses its page tables, we can mark these regions (where the guest actually stores its tables, not the private ones we use) with page protections such that the guest code will generate a fault; supervisor privilege if we are pushing all guest code to ring3, and not-present if we are pushing guest system code to ring1 should do. During the fault, our monitor will have to complete the update from the A & D bits in our private tables to those in the guest's tables.

PASS-THROUGH IO DEVICES =====

The question has been asked on several occasions, whether a guest OS can make use of native hardware. In general, the hardware that the guest OS sees, is limited to the device emulation which is offered to it, by the virtualization environment. The device emulation, in turn, makes use of the real hardware driven by the host OS, via functionality which is exported by the host OS such as libc, GUI libs, ioctl() calls, etc.

So, without any extra logic, the guest OS will not be able to use hardware which is not supported by a host OS driver and service. This is unfortunate, as it would be helpful to allow the guest OS, which has a driver for a specific piece of hardware not supported by the host OS, to drive the hardware. An example of this, would be to allow Windows as a guest OS to drive a winmodem, not supported on a Linux host OS.

This is where the concept of a pass-through device comes in. If a device is not already driven by the host OS, then there is potential for the virtualization to allow the guest OS to communicate directly with the device, within constraints of a pass-through mechanism. I believe we should explore this area at some point. We'll need to look into what is needed to accurately pass through IO reads and writes, DMA, IRQs, Plug-N-Play issues, etc. This has been done before in other projects to some degree. Anybody want to write up a section on this, and give us a background on it's use in other projects?

CUSTOM GUEST OS SPECIFIC DRIVERS =====

Due to performance considerations, lack of documentation of a specific hardware device, or development time issues, one may not want to develop emulation for a real piece of hardware. Instead, you could create a pseudo emulation, and a guest OS specific driver to communicate with it. Since you create the hardware and data exchange protocol, there is large potential for performance gains, at the expense of having to write a device driver for each guest OS.

It makes more sense to start out development, emulating a common set of devices, like the ones in bochs. But I think its worth considering the benefits derived from writing custom device emulation and associated OS drivers, soonafter. The most notable areas, where we can get some real gains, are the video, disk and network devices.

USING VIRTUALIZATION ENVIRONMENT FOR OS DEBUGGING =====

A secondary use of an accurate virtualization environment, is for debugging of a guest OS's kernel code. It is often difficult to debug kernel code, since the debugging process can be intrusive to the behaviour of the kernel.

It would be useful to have an option to conditionally compile for a debug environment, where some code in the host OS would control the virtualization environment, giving the ability to debug the guest OS in a non-intrusive manor. This would be invaluable to OS and driver design teams.

Given we employ a virtualization strategy which can virtualize arbitrary IA32 instructions, we would enjoy a great amount of flexibility. This would essentially allow us to place an infinite number of unintrusive instruction breakpoints in the VM. Or we could pass to the VM, a matrix of certain IA32 instructions that we would like to gather instrumentation data on, and collect data when these instructions are executed.

IO DEVICES FOR THE GUEST OS =====

To virtualize a complete machine, we must virtualize both the processor and the hardware devices that consitute the machine. We've talked quite a bit about virtualizing the x86 (IA32) processor. So now, we must explore issues surrounding virtualizing the hardware (IO) devices.

The nature of most devices is such that they are built to be driven by a component of an operating system, the driver. No other software must interfere with this interface, otherwise there will be conflict. As a result, we can not let the virtualized guest OS drive the same devices as are already being driven by the host OS, lest mahem will ensue.

Thus, we have no choice but to intercept all IO accesses from the CPU, and model (emulate) a complete set of devices in software. We need to pass IO reads and writes to/from the emulation, such that the IO instructions believe they are getting such information from real devices.

This is a very familiar process, as it is done in many other emulation projects. Fortunately, there is emulation for a reasonable set of IO devices already implemented and functioning in bochs. There is emulation for IDE drive, IDE ATAPI CDROM, VGA+monitor, floppy, keyboard, PICs, PITs, CMOS and RTC, limited serial port, limited NE2000 network card, etc. The components are compatible, at least to some degree with DOS, Win95, WinNT, Linux, Minux and other OSes. We may be able to share other components with various other emulation projects if need be.

Referring to the section OVERVIEW OF VIRTUALIZATION, its worth looking at the changes in context involved with virtualizing an IO operation. If the emulation of the IO device occurs in the monitor application running in the host OS context, in order to virtualize an IO operation in the guest context, the following context transitions would occur for one instruction.

- IO instruction triggers exception, guest monitor receives exception via a gate in it's IDT.
- guest monitor warps back to host monitor module.
- host monitor module transitions back to host monitor application, passing IO port information. Operation is emulated.
- monitor app calls host monitor module.
- host monitor module warps to guest monitor context, guest monitor effects the IO operation for the guest code.
- guest monitor transitions back to guest code and resumes execution.

That's 6 transitions, some of which can be very cycle expensive. For occasional IO, this is perhaps not so much of a problem. It is more of a problem for emulation of devices which receive very frequent IO requests. For example, the disk drive in IO mode would experience some serious performance penalties here. Also, the VGA (especially in planar mode) would have the same issues. The VGA frame buffer is just a memory mapped IO device. While it would be nice to just take a snapshot of the frame buffer every now and then, the latching modes make this impossible.

It's worth reiterating why we want to get back to our user mode application in the host world. Essentially, we want to have access to services provided by the host OS; access to libc, GUI, ioctl() calls and such. However, we don't need these services for a great deal of the emulation for some of the devices. So a very logical step, is split up the emulation of each device into that which can be emulated without host services, and that which can not.

For most devices, this is fortunately a very logical split. For example, let's look at the VGA emulation. That actual VGA controller has no need for host services. This can easily be moved into function provided by the monitor kernel. Periodically, we need to update our GUI window with a snapshot obtained from the VGA emulation. That update can be done back in the host application space, and is not as time critical.

Some of the devices, such as the PITs and the PICs can be totally moved into the monitor kernel functionality, since they require no host OS services. At any rate, given an IO operation can be serviced by code in the monitor kernel, here would be the set of context transitions which would occur.

- IO instruction triggers exception, guest monitor receives exception via a gate in it's IDT, guest monitor effects the IO operation for the guest code.
- guest monitor transitions back to guest code and resumes execution.

So we have trimmed down to 2 transitions. Better yet, our context switches were only "verticle" privilege level transitions. Moving "horizontal" on the diagram (between host and monitor worlds), is very cycle expensive, first because it involves a lot of context saving/restoring, and second because it mandates that we have to reload the paging register each time. Though, there may be some value in using Global pages here (persistent across page register reloads) for better performance, when this feature is available. (Global bit introduced on the Pentium Pro)

BIOS =====

We will need a reasonably complete BIOS, which is compatible with the devices we emulate. I am donating the BIOS I wrote for bochs, which is of course compatible with the device emulation from bochs, to serve this purpose. We are free to use another BIOS as well. It's best to start with a known quantity.

USING HARDWARE BREAKPOINTS IN THE GUEST OS

One of the benefits of using software breakpoints as part of our virtualization strategy, is that it leaves open the possibility of supporting hardware breakpointing in the guest OS, which can be very valuable.

Access to the hardware breakpoint registers is always virtualized in our strategies. This gives us some flexibility. We'll always know when the host attempts to change the debug registers, or read them. We are free to alter these values to something that fits our virtualization scheme. For instance, if we employ the split I&D TLB method, while we're running in the private code page, if there is a hardware breakpoint which points to a linear address in the original code page, we can temporarily change it to point into the private code page. Since we monitor out-of-page control transfers, we always have a place to synchronize the debug registers.

Also an issues, is the use of hardware breakpoints in the host OS. If during a transition from host to monitor context, we find that any hardware breakpoints are enabled, we must save the debug register contents. We must then disable any enabled ones, or replace them with values from the monitor/guest environment. A restore of such values is of course necessary on the transition back to host context.

If the host OS has not enabled any hardware breakpoints, and they're not currently in demand from the guest OS, we also have the option to do nothing on the context switch, and then dynamically save the debug registers, upon demand from the guest OS.

PROFILE FEEDBACK DATABASE ASSOCIATED WITH EACH GUEST OS

Each guest OS that you might run in the VM, has it's own set of features and resource usage. Depending on the usage of such resources, it may be possible to tweak the virtualization to do something more efficiently.

It may be therefore beneficial to store certain information about the resource usage of a given guest OS, along with it's configuration files. This would give an adaptive way to tune the VM, for the next time this guest OS is booted.

Alternatively, the user could provide info about the kind of guest OS to be run. This is a more coarse-grained and non-adaptive alternative, but perhaps effective.

PAGE CLUSTERS =====

Some of the techniques, such as the split I&D TLB technique, and the dynamic scan-before-execute technique, are discussed using a single-page oriented strategy. For example, let's look at the scan-before-execute technique. As mentioned, we always intervene when out-of-page branches are taken (and computed ones). Each such intervention results in an exception and execution of additional monitor code, and of course a corresponding performance hit.

We may find that code does not exhibit a high enough degree of locality within the constraints of a single page, that it can execute as efficiently as we'd like, due to intervention.

A natural extension to the scanning technique is to look at a cluster or series of pages as an atomic region of memory. So rather than thinking about branches out-of-page, we can look at a set of N pages as one entity and then only control static branches out of the N page region. We let intra-region branches execute without being virtualized. As a consequence, where we completely invalidate a page using the single-page strategy, we must now invalidate the page cluster, since it is an atomic structure.

There are several approaches we could take in defining what a page cluster is. A simple approach, might be to define a page cluster as a contiguous region of N pages, aligned on an N-page boundary. Effectively, this gives us a larger page size, extending it from 4096 to N*4096. Perhaps we would find that expanding the effective page size, using this technique, would solve a fair amount of our code locality performance concerns.

Taking a more dynamic slant, we could decide to start with a single page, and annex adjacent pages, up to a certain limit. For the cost of a slight bit more logic, we would prevent the inclusion of extaneous pages that have no local static branches into the current ones. The reason we don't want to include any more than necessary, is when we do need to invalidate pages, we'd like to not invalidate any more than necessary. The same reason, also puts a limitation on the number of pages we should let the page cluster grow to.

If we were really into data-flow analysis, we could dynamically maintain edge graphs of all the code branching, and create non-contiguous page clusters, accordingly. Of course, this means that part of our code page invalidation maintenance would involve an algorithm to invalidate parts of the edge graph. I'll leave this to the people with higher degrees. :^)

CONDITIONS UNDER WHICH SCAN-BEFORE-EXECUTE CAN BE ELIMINATED

=====

[I plan to fill in more here later +++]

USING PROTECTED MODE AND V86 MODE VIRTUAL INTERRUPTS

=====

[I plan to fill in more here later +++]

CAVEATS =====

Caveat #1:

If we modify fields in guest descriptors, or introduce extra descriptors to the guest descriptor tables, then the guest can conceivably see these differences with LSL, LAR, or perhaps other segment oriented instructions. For perfection, if such modifications are done, we need to virtualize such instructions.

Caveat #2:

If we want to have supervisor level code, access pages in a write-protectable way, we have to kick on the CR0.WP flag to get this effect. Otherwise supervisor code can stomp on any page regardless of it's read/write flag. This flag will already be on for host OSes which use an efficient fork() implementation, since you need it for an on-demand copy-on-write strategy. It can be easily saved/restored during the host<->monitor switch, if not.