Virtual Machines

Required reading: Disco

Overview

What is a virtual machine? IBM definition: a fully protected and isolated copy of the underlying machine's hardware.

Another view is that it provides another example of a kernel API. In contrast to other kernel APIs (unix, microkernel, and exokernel), the virtual machine operating system exports as the kernel API the processor API (e.g., the x86 interface). Thus, each program running in user space sees the services offered by a processor, and each program sees its own processor. Of course, we don't want to make a system call for each instruction, and in fact one of the main challenges in virtual machine operation systems is to design the system in such a way that the physical processor executes the virtual processor API directly, at processor speed.

Virtual machines can be useful for a number of reasons:

1. Run multiple operating systems on single piece of hardware. For example, in one process, you run Linux, and in another you run Windows/XP. If the kernel API is identical to the x86 (and faithly emulates x86 instructions, state, protection levels, page tables), then Linux and Windows/XP, the virual machine operationg system can run these *guest* operating systems without modifications.
    - Run "older" programs on the same hardware (e.g., run one x86 virtual machine in real mode to execute old DOS apps).
    - Or run applications that require different operating system.
2. Fault isolation: like processes on UNIX but more complete, because the guest operating systems runs on the virtual machine in user space. Thus, faults in the guest OS cannot effect any other software.
3. Customizing the apparent hardware: virtual machine may have different view of hardware than is physically present.
4. Simplify deployment/development of software for scalable processors (e.g., Disco).

If your operating system isn't a virtual machine operating system, what are the alternatives? Processor simulation (e.g., bochs) or binary emulation (WINE). Simulation runs instructions purely in software and is slow (e.g., 100x slow down for bochs); virtualization gets out of the way whenever possible and can be efficient.

Simulation gives portability whereas virtualization focuses on performance. However, this means that you need to model your hardware very carefully in software. Binary emulation focuses on just getting system call for a particular operating system's interface.

Binary emulation can be hard because it is targetted towards a particular operating system (and even that can change between revisions).

To provide each process with its own virtual processor that exports the same API as the physical processor, what features must the virtual machine operating system virtualize?

1. CPU: instructions -- trap all privileged instructions
2. Memory: address spaces -- map "physical" pages managed by the guest OS to *machine*pages, handle translation, etc.
3. Devices: any I/O communication needs to be trapped and passed through/handled appropriately.

The software that implements the virtualization is typically called the monitor, instead of the virtual machine operating system.

Virtual machine monitors (VMM) can be implemented in two ways:

1. Run VMM directly on hardware: like Disco.
2. Run VMM as an application (though still running as root, with integration into OS) on top of a *host* OS: like VMware. Provides additional hardware support at low development cost in VMM. Intercept CPU-level I/O requests and translate them into system calls (e.g. `read()`).

The three primary functions of a virtual machine monitor are:

- virtualize processor (CPU, memory, and devices)
- dispatch events (e.g., forward page fault trap to guest OS).
- allocate resources (e.g., divide real memory in some way between the physical memory of each guest OS).

Virtualization in detail

Memory virtualization

Understanding memory virtualization. Let's consider the MIPS example from the paper. Ideally, we'd be able to intercept and rewrite all memory address references. (e.g., by intercepting virtual memory calls). Why can't we do this on the MIPS? (There are addresses that don't go through address translation --- but we don't want the virtual machine to directly access memory!) What does Disco do to get around this problem? (Relink the kernel outside this address space.)

Having gotten around that problem, how do we handle things in general?

```
// Disco's tlb miss handler.
// Called when a memory reference for virtual adddress
// 'VA' is made, but there is not VA->MA (virtual -> machine)
```

```
// mapping in the cpu's TLB.
void tlb_miss_handler (VA)
{
  // see if we have a mapping in our "shadow" tlb (which includes
  // "main" tlb)
  tlb_entry *t = tlb_lookup (thiscpu->l2tlb, va);
  if (t && defined (thiscpu->pmap[t->pa]))   // is there a MA for this
PA?
    tlbwrite (va, thiscpu->pmap[t->pa], t->otherdata);
  else if (t)
    // get a machine page, copy physical page into, and tlbwrite
  else
    // trap to the virtual CPU/OS's handler
}

// Disco's procedure which emulates the MIPS
// instruction which writes to the tlb.
//
// VA -- virtual addresss
// PA -- physical address (NOT MA machine address!)
// otherdata -- perms and stuff
void emulate_tlbwrite_instruction (VA, PA, otherdata)
{
  tlb_insert (thiscpu->l2tlb, VA, PA, otherdata); // cache
  if (!defined (thiscpu->pmap[PA])) { // fill in pmap dynamically
    MA = allocate_machine_page ();
    thiscpu->pmap[PA] = MA; // See 4.2.2
    thiscpu->pmapbackmap[MA] = PA;
    thiscpu->memmap[MA] = VA; // See 4.2.3 (for TLB shootdowns)
  }
  tlbwrite (va, thiscpu->pmap[PA], otherdata);
}

// Disco's procedure which emulates the MIPS
// instruction which read the tlb.
tlb_entry *emulate_tlbread_instruction (VA)
{
  // Must return a TLB entry that has a "Physical" address;
  // This is recorded in our secondary TLB cache.
  // (We don't have to read from the hardware TLB since
  // all writes to the hardware TLB are mediated by Disco.
  // Thus we can always keep the l2tlb up to date.)
  return tlb_lookup (thiscpu->l2tlb, va);
}
```

CPU virtualization

Requirements:

1. Results of executing non-privileged instructions in privileged and user mode must be equivalent. (Why? B/c the virtual "privileged" system will not be running in true "privileged" mode.)
2. There must be a way to protect the VM from the real machine. (Some sort of memory protection/address translation. For fault isolation.)

3. There must be a way to detect and transfer control to the VMM when the VM tries to execute a sensitive instruction (e.g. a privileged instruction, or one that could expose the "virtualness" of the VM.) It must be possible to emulate these instructions in software. Can be classified into completely virtualizable (i.e. there are protection mechanisms that cause traps for all instructions), partly (insufficient or incomplete trap mechanisms), or not at all (e.g. no MMU).

The MIPS didn't quite meet the second criteria, as discussed above. But, it does have a supervisor mode that is between user mode and kernel mode where any privileged instruction will trap.

What might a the VMM trap handler look like?

```
void privilege_trap_handler (addr) {
  instruction, args = decode_instruction (addr)
  switch (instruction) {
  case foo:
    emulate_foo (thiscpu, args, ...);
    break;
  case bar:
    emulate_bar (thiscpu, args, ...);
    break;
  case ...:
    ...
  }
}
```

The `emulator_foo` bits will have to evaluate the state of the virtual CPU and compute the appropriate "fake" answer.

What sort of state is needed in order to appropriately emulate all of these things?

```
- all user registers
- CPU specific regs (e.g. on x86, %crN, debugging, FP...)
- page tables (or tlb)
- interrupt tables
```

This is needed for each virtual processor.

Device I/O virtualization

We intercept all communication to the I/O devices: read/writes to reserved memory addresses cause page faults into special handlers which will emulate or pass through I/O as appropriate.

In a system like Disco, the sequence would look something like:

1. VM executes instruction to access I/O
2. Trap generated by CPU (based on memory or privilege protection) transfers control to VMM.

3. VMM emulates I/O instruction, saving information about where this came from (for demultiplexing async reply from hardware later) .
4. VMM reschedules a VM.

Interrupts will require some additional work:

1. Interrupt occurs on real machine, transfering control to VMM handler.
2. VMM determines the VM that ought to receive this interrupt.
3. VMM causes a simulated interrupt to occur in the VM, and reschedules a VM.
4. VM runs its interrupt handler, which may involve other I/O instructions that need to be trapped.

The above can be slow! So sometimes you want the guest operating system to be aware that it is a guest and allow it to avoid the slow path. Special device drivers or changing instructions that would cause traps into memory read/write instructions.

Intel x86/vmware

VMware, unlike Disco, runs as an application on a guest OS and cannot modify the guest OS. Furthermore, it must virtualize the x86 instead of MIPS processor. Both of these differences make good design challenges.

The first challenge is that the monitor runs in user space, yet it must dispatch traps and it must execute privilege instructions, which both require kernel privileges. To address this challenge, the monitor downloads a piece of code, a kernel module, into the guest OS. Most modern operating systems are constructed as a core kernel, extended with downloadable kernel modules. Privileged users can insert kernel modules at run-time.

The monitor downloads a kernel module that reads the IDT, copies it, and overwrites the hard-wired entries with addresses for stubs in the just downloaded kernel module. When a trap happens, the kernel module inspects the PC, and either forwards the trap to the monitor running in user space or to the guest OS. If the trap is caused because a guest OS execute a privileged instructions, the monitor can emulate that privilege instruction by asking the kernel module to perform that instructions (perhaps after modifying the arguments to the instruction).

The second challenge is virtualizing the x86 instructions. Unfortunately, x86 doesn't meet the 3 requirements for CPU virtualization. the first two requirements above. If you run the CPU in ring 3, *most* x86 instructions will be fine, because most privileged instructions will result in a trap, which can then be forwarded to vmware for emulation. For example, consider a guest OS loading the root of a page table in CR3. This results in trap (the guest OS runs in user space), which is forwarded to the monitor, which can emulate the load to CR3 as follows:

```
// addr is a physical address
void emulate_lcr3 (thiscpu, addr)
```

```
{
  thiscpu->cr3 = addr;
  Pte *fakepdir = lookup (addr, oldcr3cache);
  if (!fakepdir) {
    fakedir = ppage_alloc ();
    store (oldcr3cache, addr, fakedir);
    // May wish to scan through supplied page directory to see if
    // we have to fix up anything in particular.
    // Exact settings will depend on how we want to handle
    // problem cases below and our own MM.
  }
  asm ("movl fakepdir,%cr3");
  // Must make sure our page fault handler is in sync with what we do
here.
}
```

To virtualize the x86, the monitor must intercept any modifications to the page table and substitute appropriate responses. And update things like the accessed/dirty bits. The monitor can arrange for this to happen by making all page table pages inaccessible so that it can emulate loads and stores to page table pages. This setup allow the monitor to virtualize the memory interface of the x86.

Unfortunately, not all instructions that must be virtualized result in traps:

- `pushf/popf`: `FL_IF` is handled different, for example. In user-mode setting FL_IF is just ignored.
- Anything (`push`, `pop`, `mov`) that reads or writes from `%cs`, which contains the privilege level.
- Setting the interrupt enable bit in EFLAGS has different semantics in user space and kernel space. In user space, it is ignored; in kernel space, the bit is set.
- And some others... (total, 17 instructions).

These instructions are unpriviliged instructions (i.e., don't cause a trap when executed by a guest OS) but expose physical processor state. These could reveal details of virtualization that should not be revealed. For example, if guest OS sets the interrupt enable bit for its virtual x86, the virtualized EFLAGS should reflect that the bit is set, even though the guest OS is running in user space.

How can we virtualize these instructions? An approach is to decode the instruction stream that is provided by the user and look for bad instructions. When we find them, replace them with an interrupt (`INT 3`) that will allow the VMM to handle it correctly. This might look something like:

```
void initcode () {
  scan_for_nonvirtual (0x7c00);
}

void scan_for_nonvirtualizable (thiscpu, startaddr) {
  addr  = startaddr;
  instr = disassemble (addr);
```

```
  while (instr is not branch or bad) {
    addr += len (instr);
    instr = disassemble (addr);
  }
  // remember that we wanted to execute this instruction.
  replace (addr, "int 3");
  record (thiscpu->rewrites, addr, instr);
}

void breakpoint_handler (tf) {
  oldinstr = lookup (thiscpu->rewrites, tf->eip);
  if (oldinstr is branch) {
    newcs:neweip = evaluate branch
    scan_for_nonvirtualizable (thiscpu, newcs:neweip)
    return;
  } else { // something non virtualizable
    // dispatch to appropriate emulation
  }
}
```

All pages must be scanned in this way. Fortunately, most pages probably are okay and don't really need any special handling so after scanning them once, we can just remember that the page is okay and let it run natively.

What if a guest OS generates instructions, writes them to memory, and then wants to execute them? We must detect self-modifying code (e.g. must simulate buffer overflow attacks correctly.) When a write to a physical page that happens to be in code segment happens, must trap the write and then rescan the affected portions of the page.

What about self-examining code? Need to protect it some how---possibly by playing tricks with instruction/data TLB caches, or introducing a private segment for code (%cs) that is different than the segment used for reads/writes (%ds).

Some Disco paper notes

Disco has some I/O specific optimizations.

- Disk reads only need to happen once and can be shared between virtual machines via copy-on-write virtual memory tricks.
- Network cards do not need to be fully virtualized --- intra VM communication doesn't need a real network card backing it.
- Special handling for NFS so that all VMs "share" a buffer cache.

Disco developers clearly had access to IRIX source code.

- Need to deal with KSEG0 segment of MIPS memory by relinking kernel at different address space.
- Ensuring page-alignment of network writes (for the purposes of doing memory map tricks.)

Performance?

- Evaluated in simulation.
- Where are the overheads? Where do they come from?
- Does it run better than NUMA IRIX?

Premise. Are virtual machine the preferred approach to extending operating systems? Have scalable multiprocessors materialized?

Related papers

Robin, John Scott, and Cynthia E. Irvine. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor." In *Proceedings of the 9th USENIX Security Symposium*. Denver, CO, August 2000.

Sugerman, Jeremy, Ganesh Venkitachalam, and Beng-Hong Lim. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor." In *Proceedings of the 2001 Usenix Annual Technical Conference.* Boston, MA, June 25-30, 2001.

Lawton, Kevin, and Drew Northup. Plex86 Virtual Machine.

Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization." SOSP '03, Bolton Landing, NY, October 19-22, 2003. (PDF)

Adams, Keith, and Ole Agesen. "A comparison of software and hardware techniques for x86 virtualizaton." ASPLOS '06, San Jose, CA, October 21-25, 2006. (PDF)