Virtual machines

Virtual machine systems can give everyone the OS (and hardware) that they want.

IBM’s VM provided an exact copy of the hardware to the user.

Advantages

You can choose your OS.

You can modify or develop new OSs without crashing machines or having to reboot.

An extra level of safety – each user’s virtual machine is completely separate from all the others.

Virtual servers (next slide)

Disadvantages

Some resources allocated to one VM can’t be shared by another. (Virtual networks)

Tricky to implement. An extra layer of complexity. Each layer can provide its own bugs.

Virtual Servers

Virtual machines are very widespread.

Many web and database servers assume that they are the only thing running on a machine. They are designed that way (is this a good idea?).

Using virtual servers means that multiple servers can be running simultaneously on the one machine.

Errors, or security problems with one server do not affect the other servers on the same machine. (In theory.)

As long as the expected loads are not going to overwhelm the machine, this is definitely a cost effective solution. Less hardware to buy and you use what you have more efficiently.

Added flexibility - VMs can be migrated from one machine to another without having to reboot.

Copy - it is trivial to create a new VM as a copy of an existing one.

Virtualization

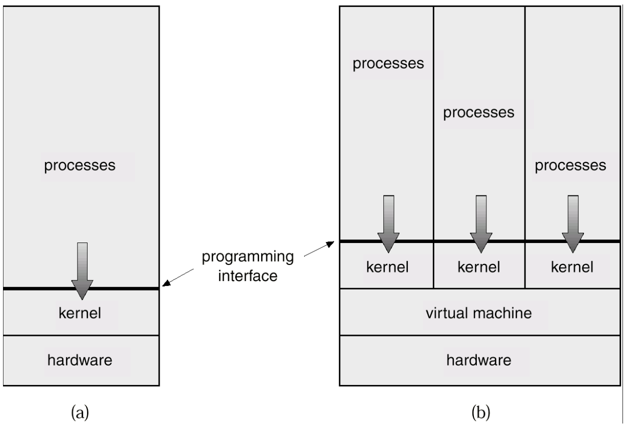
Popek and Goldberg (1974)

- Fidelity – software should run identically (except for speed) on the Virtual Machine as on the real machine.
- Performance – most instructions in a VM must run directly on the hardware (therefore at the same speed as on the real machine). Not the same as emulation.
- Safety (also known as resource control)– the Virtual Machine Monitor/Manager (VMM) is in complete control of system resources and must be safe from any actions of the VM. Also one VM must be safe from the actions of another VM.

Host OS – the operating system which the VMM is running on.

Guest OS – the operating system running inside a VM.

Design of IBM’s VM



CPU scheduling can create the appearance that users have their own processor.

Spooling and a file system can provide virtual line printers and virtual disks (minidisks).

A normal user time-sharing terminal serves as the virtual machine operator’s console.

Virtual user and kernel modes are provided.

VM did very little emulation. System calls caused traps to the VM and calls back to the correct kernel.

Only privileged instructions needed to be emulated.

Hypervisor Types

The textbook has 3 types of hypervisor (or VMMs), Popek and Goldberg had 2.

Type 0 (not universally accepted as a type) - implemented in hardware and firmware, it loads at boot time. The guest OSs load into partitions separated by the hardware. They are allocated dedicated resources e.g. processors, memory, devices. Guests OSs are native with a subset of the hardware.

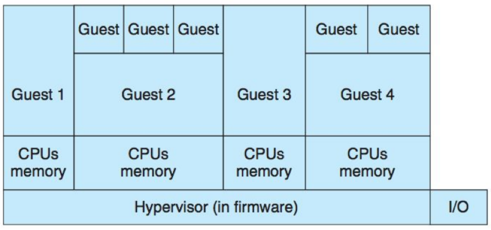


Figure 16.5 Type 0 hypervisor.

Type 1 & Type 2

Type 1

Special purpose OSs - they load at boot time and provide the optimised environment to run guest OSs. Now supported by hardware on Intel and AMD processors (see later).

Run in kernel mode.

They implement device drivers and the guests access the devices through them.

They also provide services to manage the guests - backup, monitoring. So these are the VMs used in data centres or the cloud.

e.g. VMWare's ESX, XenServer

Some "standard" OSs can also be made to run as Type 1 - e.g. Enterprise Linux, Windows Hyper-V

Type 2

These run as applications on the host OS. e.g. VMWare Workstation (or Player) and VirtualBox.

x86 Virtualization problems

- Until 2006, all x86 type CPUs had problems with classical – trap and emulate virtualization.

http://en.wikipedia.org/wiki/X86_virtualization

- Some instructions ran in both user and kernel modes (they just worked differently in kernel mode). So they were not privileged instructions and executing them did not cause a trap into the VMM.
- Some instructions allowed the program to determine if it was running in privileged mode. The kernel of a guest OS should be privileged however if it checked it would see it was in user mode, breaking the fidelity requirement.
- Even worse there were problems with protecting the page table information and keeping it all consistent.

Solutions

- User level code is fine – it will actually run in user mode and can't do anything privileged. If it tries to it will cause an exception which will be caught by the VMM and passed back to the guest OS kernel.
- So the problem only occurs when in the kernel of the guest OS.
- Binary translation -
  - Code running in kernel mode is translated at run-time into something which doesn't have these problems.

Isn't that terribly slow?

  - The translation is very simple (and hence efficient).
  - Only translates code which is actually run.
  - Much of the code is exactly the same as the original.
  - The translated code is cached and reused.
  - Uses all sorts of tricks to speed up emulation.

- Performs very well compared to true hardware virtualization – which struggles with page-table modifications.

Hardware Virtualization (x86)

Both Intel and AMD have developed their own solutions to deal with virtualization.

Intel VT and AMD-V – an extra high privilege area for the VMM. OS still runs in ring 0 (kernel mode).

Hardware transitions from ring 0 to the VMM.

Processor state is maintained for each guest OS (and the VMM) in separate address spaces.

AMD-V – included tagged translation lookaside buffers (so virtual memory didn't have a hit when changing virtual machines)

Both AMD and Intel processors now do Second Level Address Translation (SLAT)

- determine the guest physical address from the guest virtual address using hardware
- then turn the guest physical address into the host physical address also using hardware

OS level virtualization

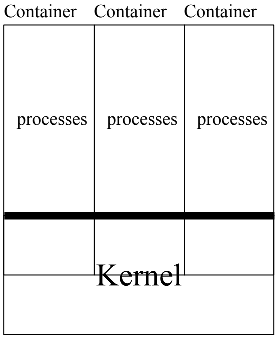
If virtualizing servers we can often use the same OS. This means we virtualize less.

Containers look like servers – they can be rebooted separately, have their own IP addresses, root, programs etc.

But they all use the same underlying kernel.

And they are still separate from each other.

e.g. Parallels Virtuozzo and OpenVZ - see http://en.wikipedia.org/wiki/OpenVZ



More VM styles

Paravirtualization - Xen – software approach. Requires modifications to the OS source code to use the Xen layer. e.g. To read from a file the guest directly calls host read routines.

Application virtualization - an application runs on a layer which provides the resources it needs even though it may be running on a different OS e.g. Wine or running programs from old versions of Windows on a newer one.

Programming-Environment Virtualization, Java VM & CLR/Mono

- Implement a different architecture on top of any hardware/OS.
- Programs are compiled to the Java VM or CLR architecture then run by either compiling or interpreting that code.
- Earlier versions – late 70's UCSD Pascal system.

C and OS implementations

Why has C been the language of choice for most operating system implementations?

That is what it was designed for:

- Ken Thompson and Dennis Ritchie converted UNIX from assembly language to C to provide portability.

Close to the hardware.

- Low-level access to memory
- Maps easily to machine instructions
- Easy to inline assembly language code (depends on the compiler)

Has small requirements for runtime support

Sometimes referred to as a high-level assembler.

Direct access to memory

C pointers can have integers assigned to them

- This means that actual addresses can be stored in a pointer and then used to access that address
- Memory mapped devices can then be controlled directly from normal C.

e.g.

```
#include <stdio.h>
#include <stdlib.h>

#define ADDRESS 0x7fff52cb08a0

int main(void) {
      long number = 1234;
      long *pointer = &number;

      printf("number is %ld\n", number);
      printf("The value at %p is %ld.\n", pointer, *pointer);
      *pointer = 42;
      printf("number is %ld\n", number);

      // And I can do the same thing with any address
     // it could cause a segmentation error
      // since MacOS now uses ASLR
      pointer = (long *)ADDRESS;
      printf("The value at %p is %ld.\n", (void *)ADDRESS, *pointer);
      return EXIT_SUCCESS;
}
```

Pointer arithmetic gave fast access to elements in arrays or structs

Accessing registers

You can use the “register” storage class specifier to say that a variable should be kept in a processor register. e.g.

```
register long number = 1234;
```

However this does not guarantee that a value is stored in a register, it depends on the number of available registers.

Also compilers do a really good job of optimising register usage and it is not usually a good idea for a programmer to worry about this level of optimisation.

Memory mapped registers can be accessed directly using pointer manipulation as on the previous slide.

Volatile

Volatile is a storage class qualifier

e.g.

```
volatile unsigned char *reg;
```

This means the variable may change in a non-local way, in other words there is no way the compiler could possibly know whether the value has changed or not between references.

So the compiler is not allowed to optimise accesses. Every single read must go back to the main store to retrieve the current value.

- memory mapped device registers
- values modified in interrupt routines
- values modified in another thread

```
unsigned char *reg = (unsigned char *)0x1000;
while (*reg) {}
```

This looks like it will always repeat or never repeat depending on the initial value stored in 0x1000.

Memory management

All local variables disappear when functions are returned from.

- Space is allocated on the stack for the variables in each function invocation.
- There is a limit to the size of the stack (especially for threads as each thread needs its own stack)

Areas of static memory

- Global variables
- Static variables - can be static in a file or in a function.

```
static int x;
```

If in a function it maintains its value even when the function is returned from.

The advantage of static memory is that it is allocated at compile time and hence has no allocation overhead at run time.

The disadvantage is that it cannot easily be released.

Dynamic Memory

C requires explicit control of dynamic memory.

This is suitable for OS programming as there is no garbage collection available.

- garbage collection adds a layer of complexity and unpredictability to the programming environment
- this is important in small systems - such as embedded systems (or phones?)
- especially important in real-time systems

To allocate memory we use malloc (or calloc or valloc etc.).

```
struct thread *thread;
if ((thread = malloc(sizeof(struct thread))) == NULL) {
     perror("allocating thread");
     exit(EXIT_FAILURE);
}
```

To deallocate memory we use free.

```
free(thread);
```

How does free know how much memory to release?

Inline assembly

This is both compiler and system dependent.

```
// store current stack pointer
int savedSP;
asm("movl %%esp, %0\n" :"=r"(savedSP));
// change to the top of newThread's stack
char *tos = stack + SIGSTKSZ - 16;
asm("movl %0, %%esp\n" : : "r"(tos));
associateStack();
// restore stack pointer
asm("movl %0, %%esp\n" : : "r"(savedSP));
```

Example Windows Code

From the Windows Research Kernel (basically Windows Server 2003, also the same as XP)

```
//
// Create the process ID
//

CidEntry.Object = Process;
CidEntry.GrantedAccess = 0;
Process->UniqueProcessId = ExCreateHandle (PspCidTable, &CidEntry);
if (Process->UniqueProcessId == NULL) {
    Status = STATUS_INSUFFICIENT_RESOURCES;
    goto exit_and_deref;
}

ExSetHandleTableOwner (Process->ObjectTable,
                       Process->UniqueProcessId);
```

Example Linux Code

```
/*
 * When we die, we re-parent all our children.
 * Try to give them to another thread in our thread
 * group, and if no such member exists, give it to
 * the child reaper process (ie "init") in our pid
 * space.
 */
static struct task_struct *find_new_reaper(struct task_struct
*father)
{
    struct pid_namespace *pid_ns = task_active_pid_ns(father);
    struct task_struct *thread;

    thread = father;
    while_each_thread(father, thread) {
        if (thread->flags & PF_EXITING)
            continue;
        if (unlikely(pid_ns->child_reaper == father))
            pid_ns->child_reaper = thread;
        return thread;
    }

    if (unlikely(pid_ns->child_reaper == father)) {
        write_unlock_irq(&tasklist_lock);
        if (unlikely(pid_ns == &init_pid_ns))
            panic("Attempted to kill init!");

        zap_pid_ns_processes(pid_ns);
        write_lock_irq(&tasklist_lock);
        /*
         * We can not clear ->child_reaper or leave it alone.
         * There may by stealth EXIT_DEAD tasks on ->children,
         * forget_original_parent() must move them somewhere.
         */
        pid_ns->child_reaper = init_pid_ns.child_reaper;
    }

    return pid_ns->child_reaper;
}
```

Running commands from a C program

Because of the large number of standard useful commands in Unix it is helpful to be able to use them from inside C programs.

Type `man system`

e.g.

```
system(“ps -x”);
```

This creates a shell and gets it to execute the command.

If you need to use the output (or provide input) in your program you need to use the `popen` C function.

Alternatives

C++ - Some operating systems are written in C++. Sometimes with a small section in C.

Objective C - MacOS is written in C with Objective-C on top

Java - Some operating systems are written largely (but not exclusively) in Java

Assembly - the original operating system implementation language

Python

But we are using Python.

Why?

It is easier to learn

It has very convenient access to Unix OS calls

It is fun

Accessing Unix system calls

```
import os
reader, writer = os.pipe()
child_pid = os.fork()

import sys
sys.stdout.write(‘hi\n’)
file = open(‘output’, ‘w’)
sys.stdout = file
print(‘Where did this go?’)
```

Before the next lecture

Read textbook sections

3.1 Process Concept

Threads

4.1 Overview

4.2 Multicore Programming

4.3 Multithreading Models

4.4 Thread Libraries

4.7 Operating-System Examples