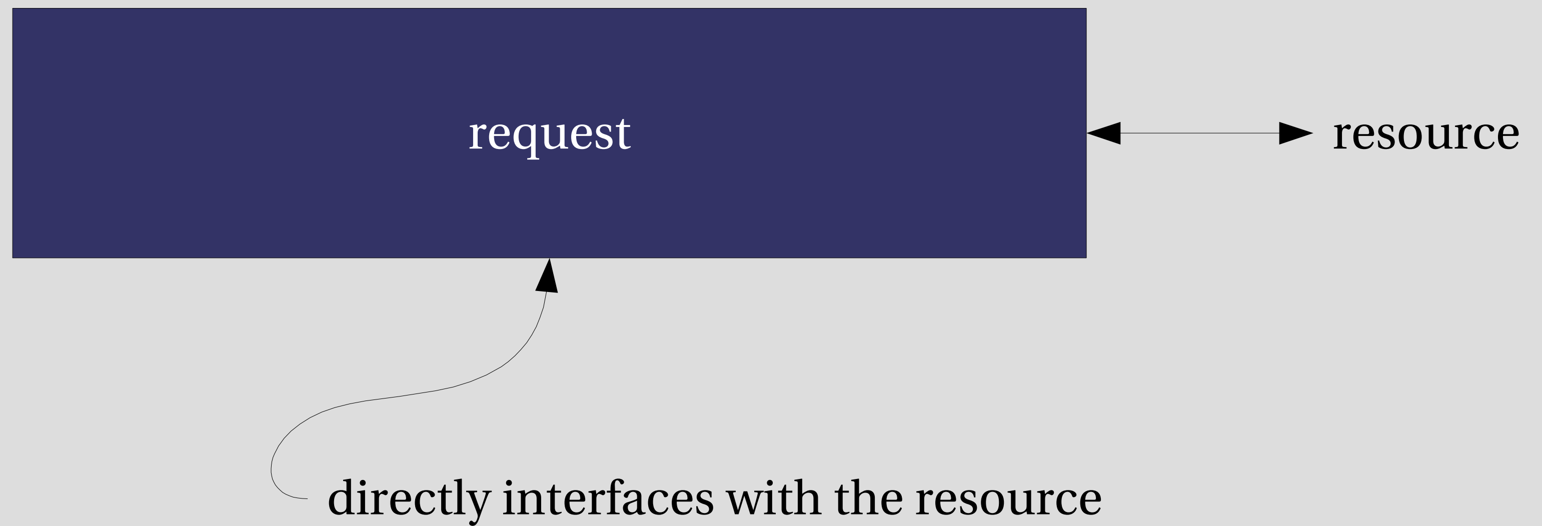
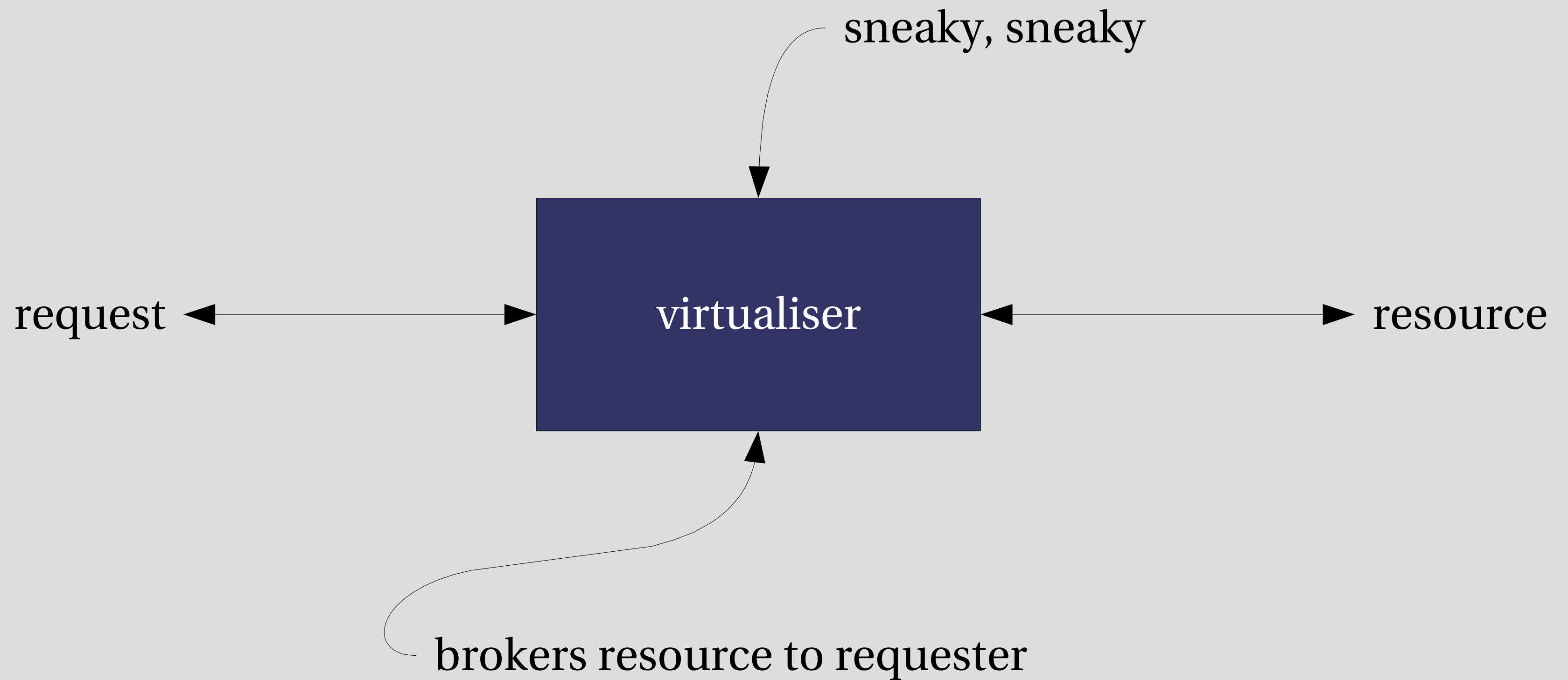
Virtualisation

Professor: Guntis Barzdins
Assistant: Kristaps Dzonsons

base case



virtualised case



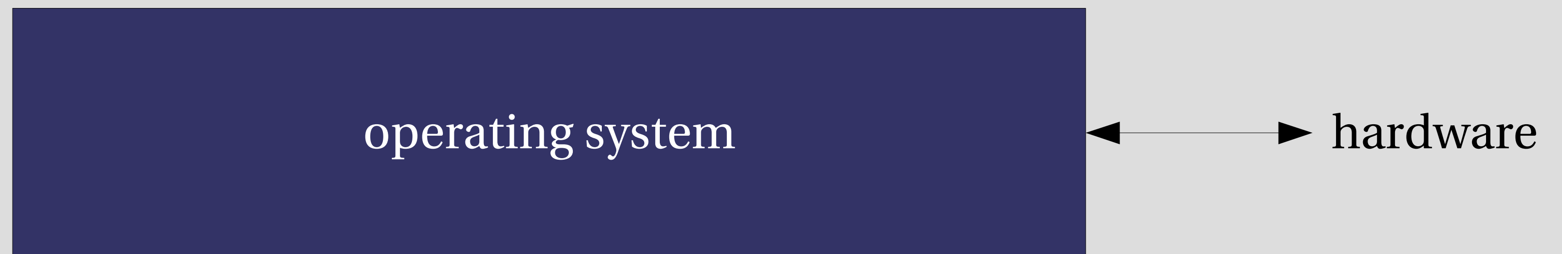
generic terminology

- virtualisation: a "middle-man" to servicing a resource request
- this lecture focusses on virtualisation as regards computer operating systems

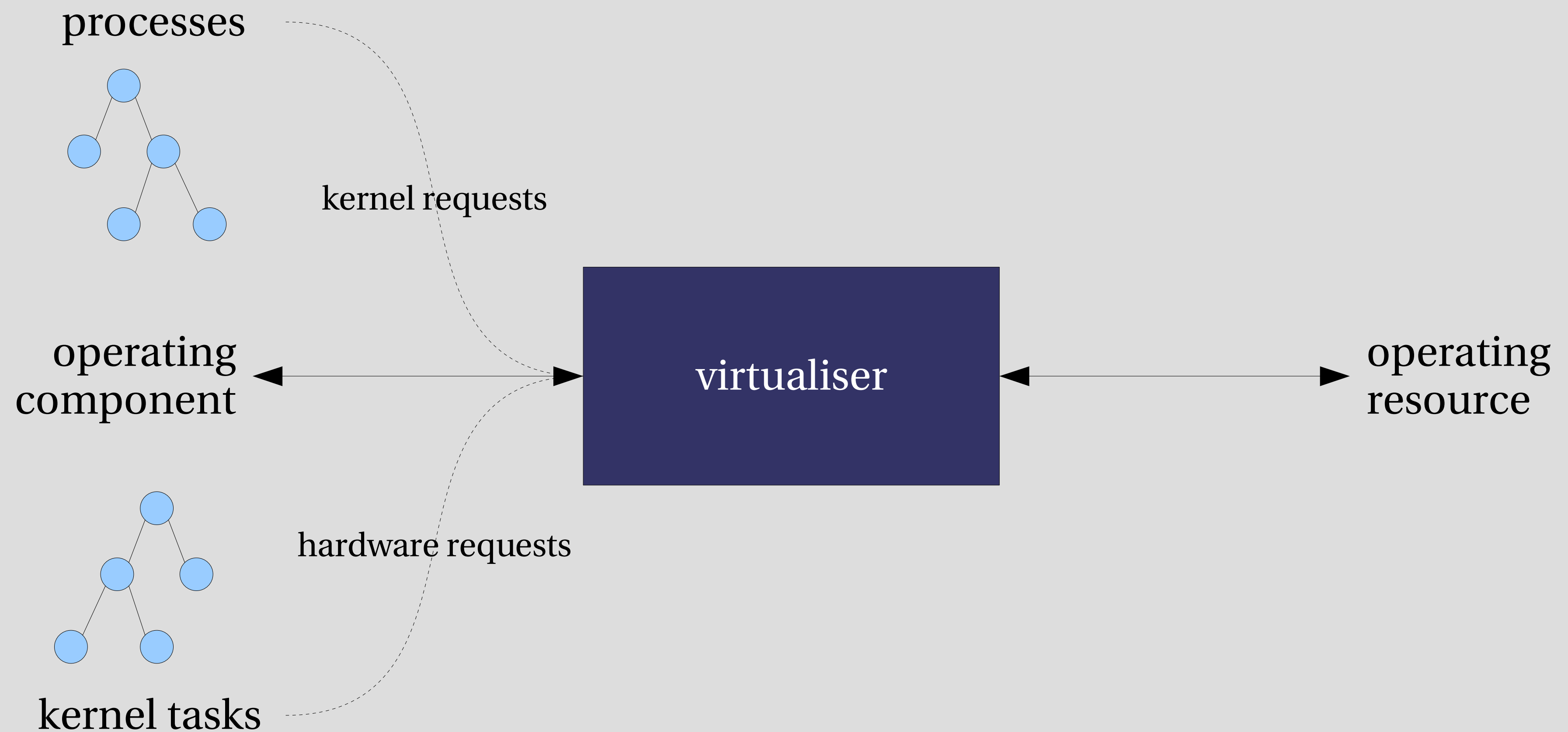
specific terminology

- operating resources
  - memory, file-system, I/O, ...
- operating instance
  - user-land processes
- operating system
  - operating instance and kernel
- operating component
  - an operating system or instance
- virtualisation
  - to broker operating resource requests from an operating component

base case revisited



virtualised case revisited

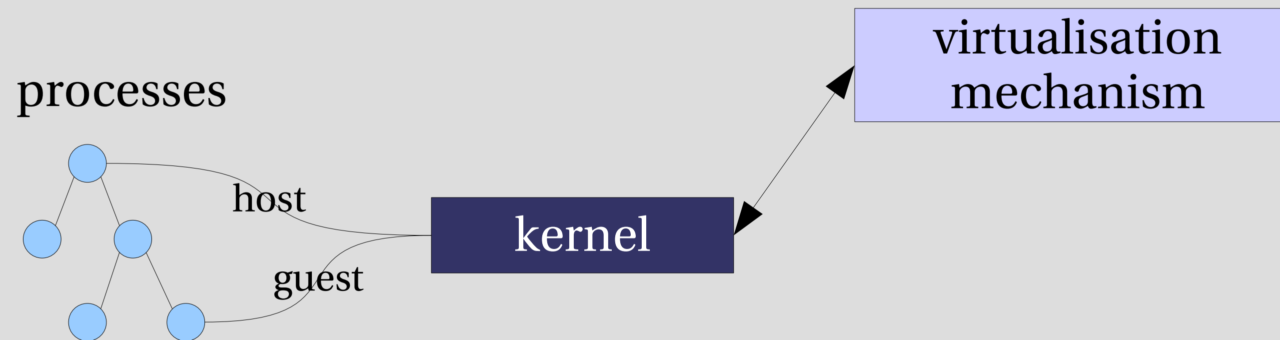


taxonomy

- "operating instance" virtualisers
  - Linux Virtuozzo
  - Linux VServer
  - Solaris Zones
  - FreeBSD Jails
- "operating system" virtualisers
  - bochs (full)
  - QEMU (full)
  - kQEMU (assisted)
  - KVM (assisted)
  - Xen (para)

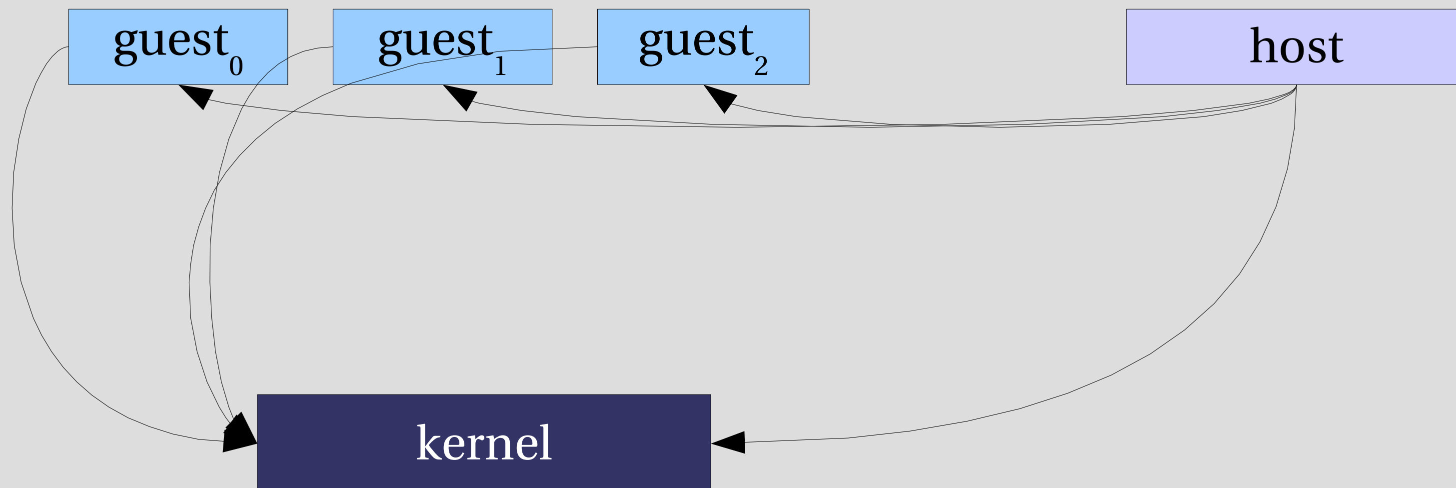
taxonomy

- instance virtualisers



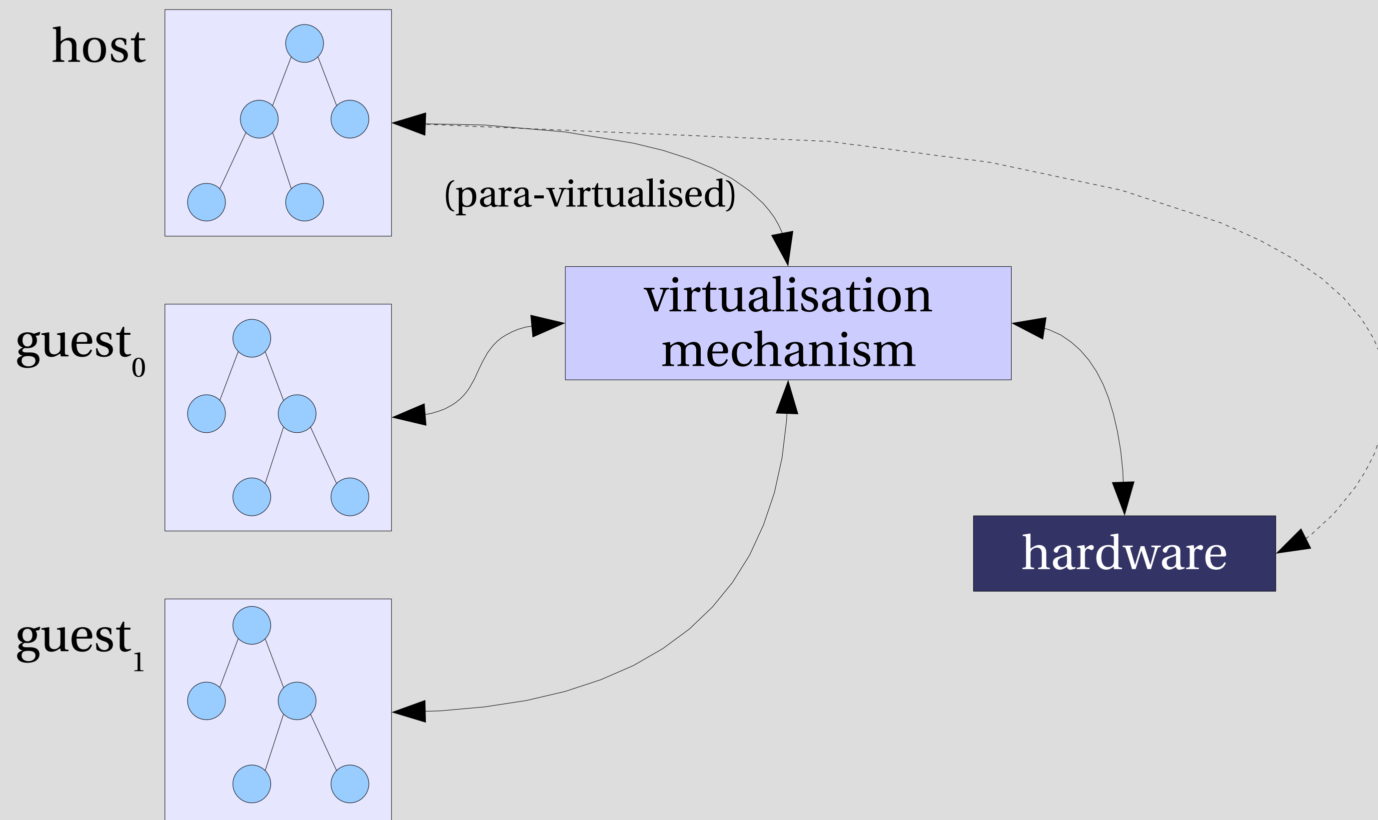
taxonomy

- instance virtualisers



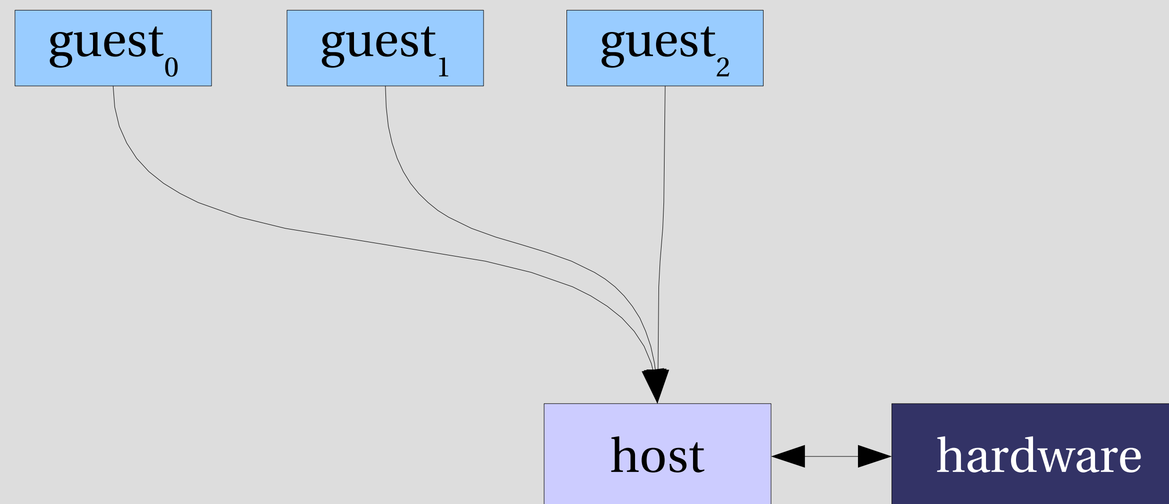
taxonomy

- system virtualisers



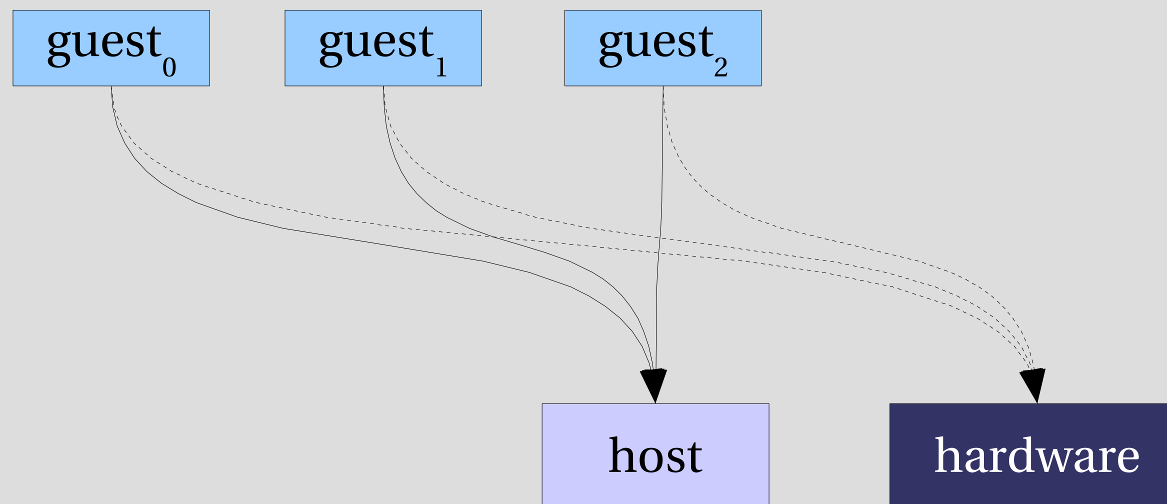
taxonomy

- system virtualisers (full)



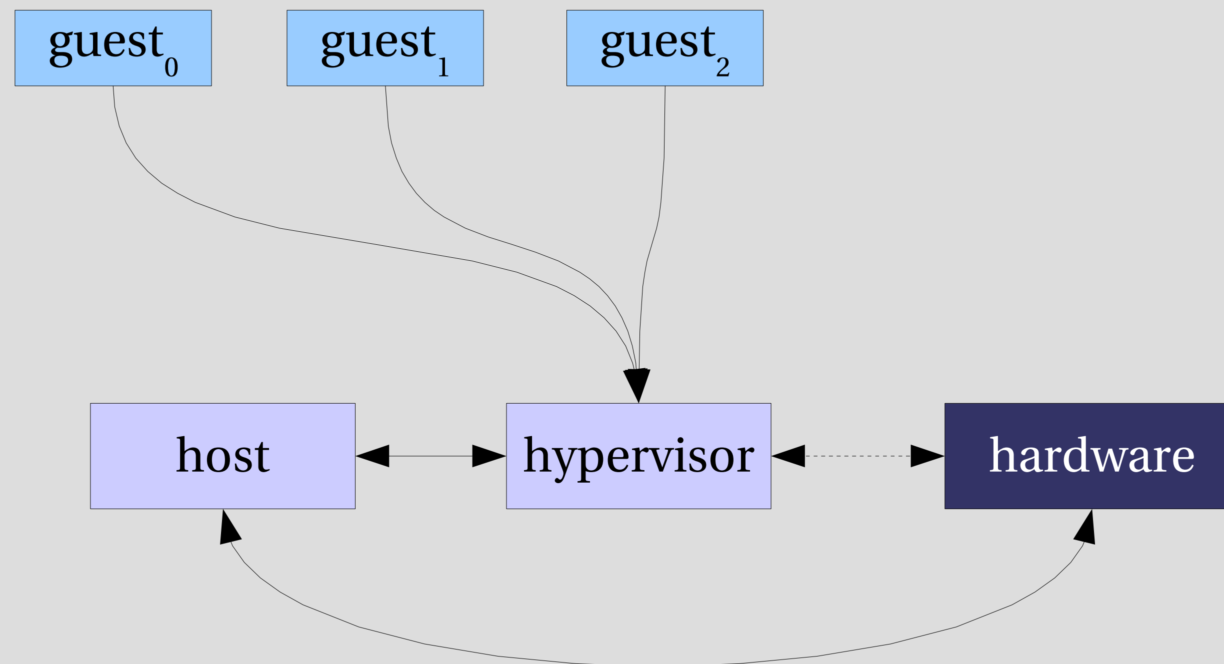
taxonomy

- system virtualisers (assisted)



taxonomy

- system virtualisers (para)



ISA formalisation

- Popek and Goldberg virtualisation (1974)
  - defines the virtualisation capabilities of an instruction set
  - equivalence: behaviour under VMM (virtual machine monitor) must be equivalent to that of the host
  - resource control: VMM must be in complete control of its virtualised resource access
  - efficiency: statistically-dominant fraction of instructions must be executed without VMM

ISA formalisation

- ISA classified in three groups:
  - privileged: instructions that trap *iff* not executing in system mode
  - control-sensitive: instructions that change the system environment
  - behaviour-sensitive: instructions depending on the state of the system environment

ISA formalisation

- Popek/Goldberg conditions
  - set of sensitive instructions is a subset of privileged instructions (Theorem 1)
  - if a machine *m* satisfies Theorem 1, and a VMM may be constructed for *m* without timing dependencies, then *m* is recursively virtualisable (Theorem 2)

formalisation

- ISA survey regarding Popek/Goldberg
  - IA-32: no (17 violations: `CALL`, `JMP`, `INT`, ...)
  - IA-64: no (see IA-32)
  - System/370: yes
  - PDP-10: no (4 violations: `JSP`, `JSR`, ...)

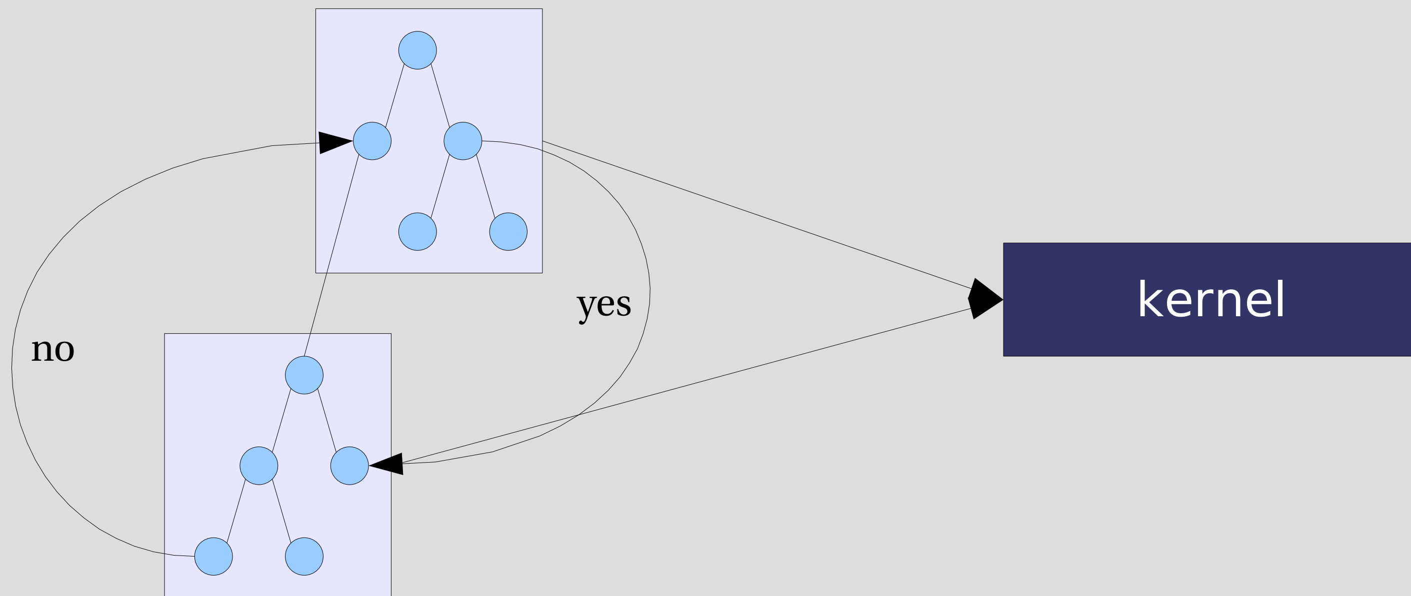
instance virtualisers

- What and how are instance virtualisers?

instance virtualisers

- virtualisation target
  - containers have multiple segmented operating environments: processes and their operating resources do not overlap between guest environments
  - environments share a single kernel image
  - the host environment may "see" guests, while guests are segmented from one another
  - containers are thus restricted to homogeneous environments (barring "compat")

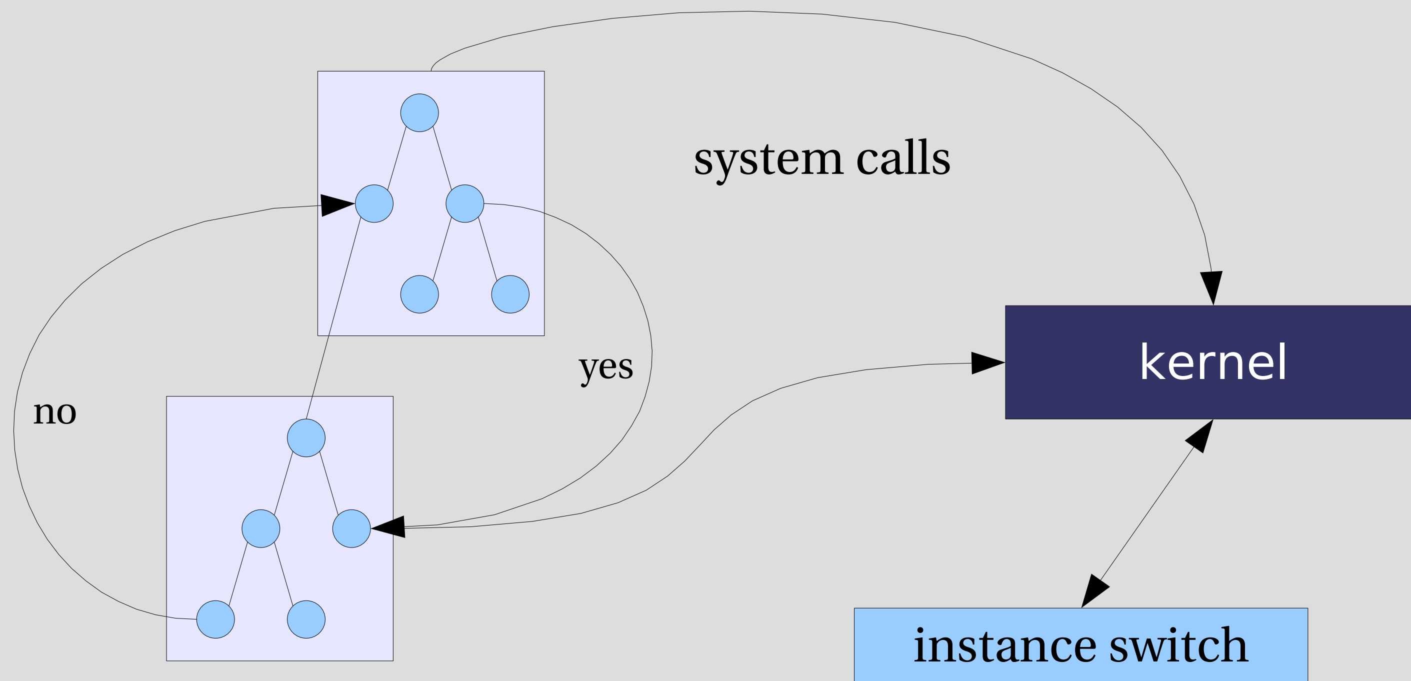
instance virtualisers



instance virtualisers

- virtualisation implementation
  - the kernel keeps track of multiple operating instances and switches when shared resources are requested
  - there is usually a host/guest hierarchy allowing the host to arbitrarily effect guest instances, but not vice-versa

instance virtualisers



case study: FreeBSD jail

- consider the instance virtualisation provided by FreeBSD's jail
- this references FreeBSD 6.2-RELEASE

FreeBSD jail implementation

1. process calls `gethostname(3)`
2. `gethostname(3)` calls `sysctl(3)`
3. `sysctl(3)` calls `__sysctl`
4. `__sysctl` triggers `sys___sysctl`
5. `sys___sysctl` switches on `KERN_HOSTNAME`
6. function verifies whether calling process is jailed, and if so, returns the jail's hostname

FreeBSD jail implementation

- sys/kern/kern_sysctl.c

```
1009    int
1010    kernel_sysctl(struct thread *td, int *name, u_int namelen, void *old,
1011        size_t *oldlenp, void *new, size_t newlen, size_t *retval, int
   flags)
1012    {
...
1039        SYSCTL_LOCK();
1040
1041        error = sysctl_root(0, name, namelen, &req);
...
1057        return (error);
1058    }
```

FreeBSD jail implementation

- sys/kern/kern_sysctl.c

```
1218    static int
1219    sysctl_root(SYSCTL_HANDLER_ARGS)
1220    {
...
1224        error = sysctl_find_oid(arg1, arg2, &oid, &indx, req);
...
1281        error = oid->oid_handler(oid, arg1, arg2, req);
1282
1283        return (error);
1284    }
```

- sys/sys/sysctl.h

```
153     struct sysctl_oid {
...
161         int         (*oid_handler)(SYSCTL_HANDLER_ARGS);
...
165     };
```

FreeBSD jail implementation

- sys/kern/kern_mib.c

```
231     SYSCTL_PROC(_kern, KERN_HOSTNAME, hostname,
232         CTLTYPE_STRING|CTLFLAGS_RW|CTLFLAGS_PRISON,
233         0, 0, sysctl_hostname, "A", "Hostname");

193     static int
194     sysctl_hostname(SYSCTL_HANDLER_ARGS)
195     {
...
200         pr = req->td->td_ucred->cr_prison;
201         if (pr != NULL) {
202             if (!jail_set_hostname_allowed && req->newptr)
203                 return (EPERM);
...
216             if (req->newptr != NULL && error == 0) {
...
221                 mtx_lock(&ptr->pr_mtx);
222                 bcopy(tmphostname, pr->pr_host, MAXHOSTNAMELEN);
223                 mtx_unlock(&pr->pr_mtx);
224             }
...
229         }
```

FreeBSD jail implementation

- sys/sys/jail.h

```
64      struct prison {
65          LIST_ENTRY(prison) pr_list;
66          int                 pr_id;
67          int                 pr_ref;
68          char                pr_path[MAXPATHLEN];
69          struct vnode        *pr_root;
70          char                pr_host[MAXHOSTNAMELEN];
71          u_int32_t           pr_ip;
72          void                *pr_linux;
73          int                 pr_securelevel;
74          struct task         pr_task;
75          struct mtx          pr_mtx;
76      };
```

instance virtualisers: cons

- cons
  - complexity: *all* possible overlap between operating instances must be specifically handled by the kernel, including (arguably most importantly) devices
  - virtualised systems must be the same as the hosting system (barring "compat" situations)

instance virtualisers: pros

- pros
  - high-speed: unless devices are inefficiently virtualised, operations proceed at native speed
  - minimal space overhead: only one kernel is required, which manages all virtualisation operations without extra over-head

instance virtualisers: cons

1. process calls `open(2)` with `/dev/cd0a`
2. process calls `write(2)` to `/dev/cd0a`
3. kernel drops into device write routine
4. unless device driver specifically handles virtualisation, the call is unrestricted

instance virtualisers: pros

- efficiency: maximum number of FreeBSD jails: approximately 64 000
- efficiency: maximum number of Solaris zones: approximately 9 000
- efficiency: maximum number of Virtuozzo images: approximately 320

system virtualisers

- What and how are system virtualisers?
  - What and how are full virtualisers?
  - What and how are assisted virtualisers?
  - What and how are para-virtualisers?

system virtualisers

- virtualisation target
  - resource requests from a guest kernel are intercepted by the virtualiser
  - these requests are specially brokered by the virtualising agent
  - since this is an expensive process (as kernels are generally in high-speed contact with hardware), there are a number of hardware facilities to optimise the process, and a number of strategies that mingle hardware and software brokerage

system virtualisers

- full virtualisation
  - image is virtualised fully by software
- assisted virtualisation
  - some operations are virtualised in software, some in hardware
- para-virtualisation
  - like assisted virtualisation, but requires re-writing of the image to "play nicely" with the virtualiser

case study

- consider the full system virtualisation provided by bochs
- this targets bochs-20070929 (2.3.5)

bochs implementation

1. `bochs(1)` called with a BIOS, boot device, ...
2. emulator memory allocated and initialised
3. BIOS (128 KB default) loaded into memory ROM arena and configured
4. BIOS executed (`0xe000` default)
5. CPU instructions translated by emulator into configured execution environment

bochs implementation

```
% cat .bochsrc
ata0-master: type=disk, path="30M.sample", cylinders=615, heads=6, spt=17
boot: disk
%
% bochs
```

instructs default BIOS to boot from given disc image

default BIOS (non-VGA) at 0xe000

bochs implementation

1. deal with asynchronous events
2. previous instruction, hardware interrupts
3. external interrupts
4. get an instruction
5. execute
6. update clock

bochs implementation

- cpu/cpu.cc (loop prologue)

```
197 void BX_CPU_C::cpu_loop(Bit32u max_instr_count)
198 {
...
209   if (setjmp(BX_CPU_THIS_PTR jmp_buf_env))
210   {
211      // only from exception function we can get here ...
212      BX_INSTR_NEW_INSTRUCTION(BX_CPU_ID);
...
222   }
...
238   BX_CPU_THIS_PTR prev_eip = RIP;
239   BX_CPU_THIS_PTR prev_esp = RSP;
240   BX_CPU_THIS_PTR EXT = 0;
241   BX_CPU_THIS_PTR errorno = 0;
242
243   while (1) {
...
306   }
307 }
```

set simulated CPU exception return point

configure stack boundaries and instruction pointer

bochs implementation

- cpu/cpu.cc (asynchronous events)

```
197 void BX_CPU_C::cpu_loop(Bit32u max_instr_count)
198 {
...
243   while (1) {
...
248     if (BX_CPU_THIS_PTR async_event) {
249       if (handleAsyncEvent()) {
250         // If request to return to caller ASAP.
251         return;
252       }
253     }
...
306   }
307 }
```

bochs implementation

- cpu/cpu.cc (asynchronous events)

```
413 unsigned BX_CPU_C::handleAsyncEvent(void)
414 {
...
525    else if (BX_CPU_INTR && BX_CPU_THIS_PTR get_IF() && BX_DBG_ASYNC_INTR)
526    {
...
537        vector = DEV_pic_iac(); // may set INTR with next interrupt
...
539        BX_CPU_THIS_PTR errorno = 0;
540        BX_CPU_THIS_PTR EXT = 1; /* external event */
541        BX_INSTR_HWINTERRUPT(BX_CPU_ID, vector,
542        BX_CPU_THIS_PTR sregs[BX_SEG_REG_CS].selector.value, RIP);
543        interrupt(vector, 0, 0, 0);
...
549        BX_CPU_THIS_PTR prev_eip = RIP; // commit new RIP
550        BX_CPU_THIS_PTR prev_esp = RSP; // commit new RSP
551        BX_CPU_THIS_PTR EXT = 0;
552        BX_CPU_THIS_PTR errorno = 0;
...
637  }
```

bochs implementation

- cpu/cpu.cc (fetch instructions)

```
197 void BX_CPU_C::cpu_loop(Bit32u max_instr_count)
198 {
...
243   while (1) {
...
263     bxInstruction_c *i = fetchInstruction(&iStorage, eipBiased);
...
306   }
307 }
```

```
108 BX_CPP_INLINE bxInstruction_c* BX_CPU_C::fetchInstruction
          (bxInstruction_c *iStorage, bx_address eipBased)
109 {
...
144     Bit8u *fetchPtr = BX_CPU_THIS_PTR eipFetchPtr + eipBased;
...
157     ret = fetchDecode32(fetchPtr, i, maxFetch);
...
127 }
```

bochs implementation

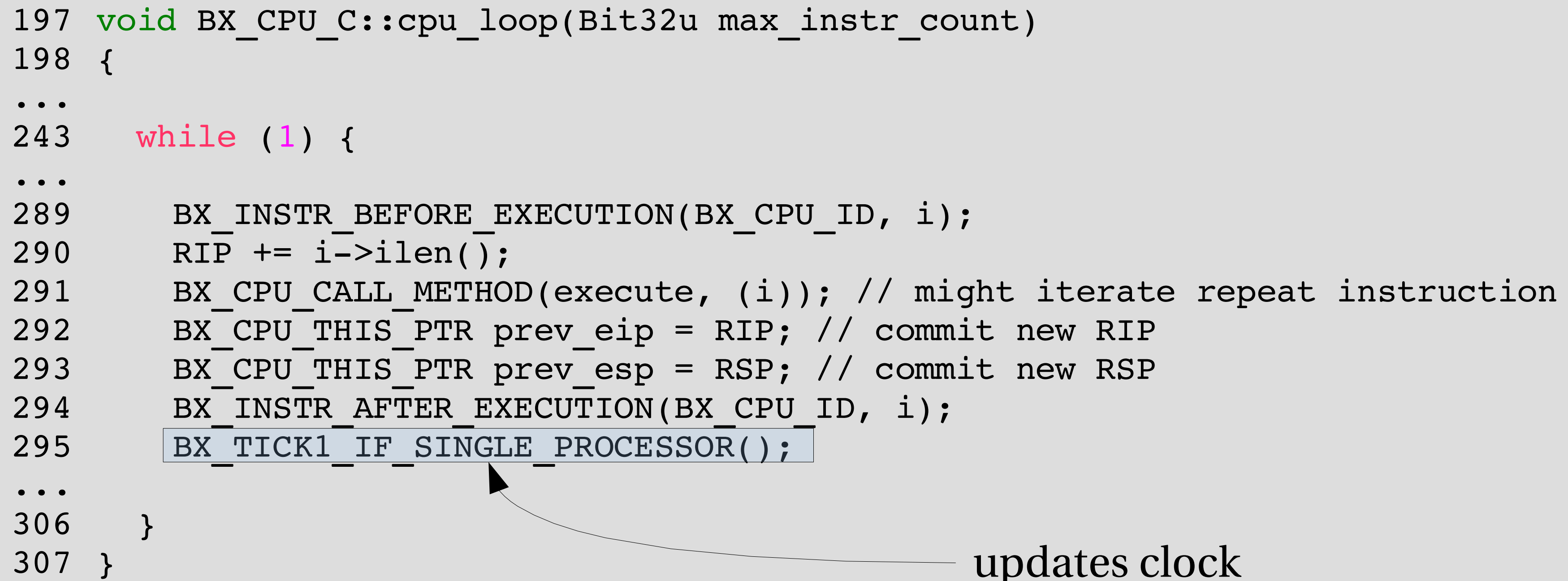
- cpu/fetchdecode.cc (fetch instructions)

```
408     static const BxOpcodeInfo_t BxOpcodeInfo[512*2] = {
...
458         /* 30 */ { BxAnother | BxLockable, &BX_CPU_C::XOR_EbGb },
459         /* 31 */ { BxAnother | BxLockable, &BX_CPU_C::XOR_EwGw },
...
1524    };

1526    unsigned
1527    BX_CPU_C::fetchDecode32(Bit8u *iptr, bxInstruction_c *instruction,
               unsigned remain)
1528    {
...
1878        instruction->execute = BxOpcodeInfo[b1+offset].ExecutePtr;
1879        instruction->IxForm.opcodeReg = b1 & 7;
...
2018    }
```

bochs implementation

- cpu/cpu.cc (execute instructions)



bochs implementation

- cpu/fetchdecode.cc (execute instructions)

```
34      void BX_CPU_C::XOR_EbGb(bxInstruction_c *i)
35      {
36          Bit8u op2, op1, result;
37
38          op2 = BX_READ_8BIT_REGx(i->nnn(),i->extend8bitL());
39
40          if (i->modC0()) {
41              op1 = BX_READ_8BIT_REGx(i->rm(),i->extend8bitL());
42              result = op1 ^ op2;
43              BX_WRITE_8BIT_REGx(i->rm(), i->extend8bitL(), result);
44          }
45          else {
46              read_RMW_virtual_byte(i->seg(), RMAddr(i), &op1);
47              result = op1 ^ op2;
48              write_RMW_virtual_byte(result);
49          }
50
51          SET_FLAGS_OSZAPC_RESULT_8(result, BX_INSTR_LOGIC8);
52      }
```

full system virtualisation

- cons
  - full system virtualisation is quite slow
  - several areas, like memory management and I/O device management, have non-trivial implementation strategies

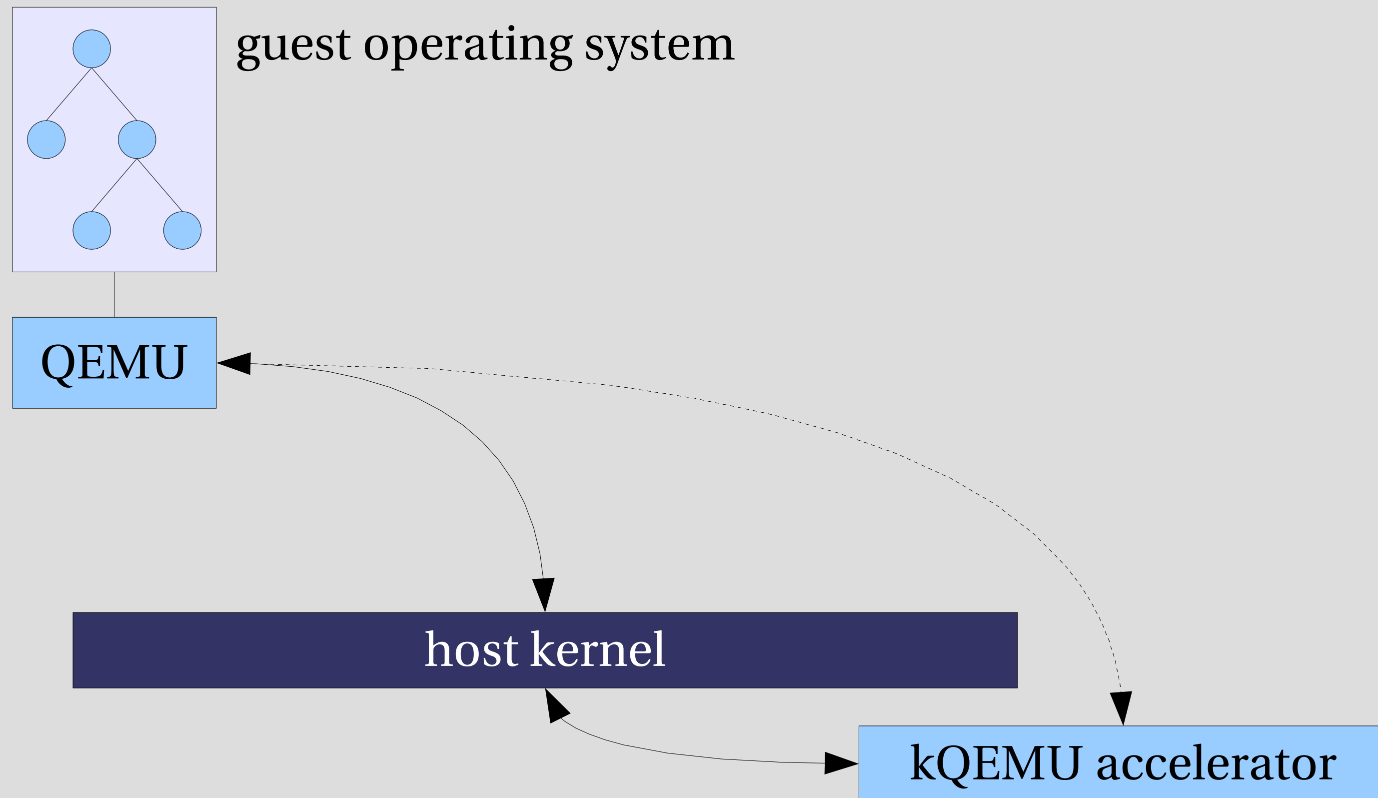
full system virtualisation

- pros
  - excellent for debugging: a full view of the system run-time down to the register file, pending I/O requests, and so on
  - if properly designed, executing systems are fully sandboxed

case study

- consider the assisted system virtualisation provided by QEMU and kQEMU
- this targets qemu-0.9.0
- this also targets kqemu-1.3.0pre11

QEMU implementation



QEMU implementation

- QEMU vs. bochs
  - dynamic compilation: code is translated while the system is executing instructions
  - QEMU supports an in-kernel extension, kQEMU, that significantly speeds up virtual system run-time
  - QEMU is *significantly* more complicated than bochs: supports many processors, has considerable optimisations, etc.

QEMU implementation

- dynamic compilation
  - instructions are cached and pre-translated up to branch instructions ("translation blocks")
  - translation occurs with `dyngen`, a special tool that creates abridged code sequences
  - translation blocks are concatenated into a single execution sequence

QEMU implementation

- instructions are broken into smaller chunks (micro-ops) by the instruction translator

PPC assembly snippet:

```
0001        addi    r1,r1,-16        # r1 -= 16
```

modified PPC assembly snippet:

```
0001        movl    t0,r1
0002        addl    t0,-16
0003        movl    r1,t0
```

QEMU implementation

- modified "micro-ops" are implemented in C code in the QEMU sources

```
001    void op_movl_T0_r1(void)
002    {
003        T0 = env->regs[1];
004    }


005    extern int __op_param1;
006    void op_addl_T0_im(void)
007    {
008        T0 = T0 + ((long)(__op_param1));
009    }
```

QEMU implementation

- micro-ops in the translation buffer are concatenated into an executable buffer

```
001     for  (;;)
002         switch (*opc_ptr++) {
...
004         case INDEX_op_movl_T0_r1: {
005             extern void op_movl_T0_r1();
006             memcpy(gen_code_ptr, (char *)&op_movl_T0_r1 + 0, 3);
007             gen_code_ptr += 3;
008             break;
009             }
010         case INDEX_op_addl_T0_im: {
011             long param1;
012             extern void op_addl_T0_im();
013             memcpy(gen_code_ptr, (char *)&op_addl_T0_r1 + 0, 6);
014             param1 = *opparam_ptr++;
015             *(uint32_t *)(gen_code_ptr + 2) = param1;
016             gen_code_ptr += 6;
017             break;
018             }
019         }
```

QEMU implementation

– resulting in the following run-time assembled machine code

```
001    # movl_T0_r1
002    # ebx = env->regs[1]
003    mov     0x4(%ebp), %ebx
004    # addl_T0_im -16
005    # ebx = ebx - 16
006    add     $0xfffffff0, %ebx
007    # movl_r1_T0
008    # env->regs[1] = ebx
009    mov     %ebx, 0x4(%ebp)
```

QEMU implementation

- handling self-modifying code?
  - many languages allow for code to change itself during run-time
  - by marking translated pages as read-only (with `mprotect(2)`), QEMU can handle `SIGSEGV` and `SIGBUS` and flush the translation buffer when pages are modified

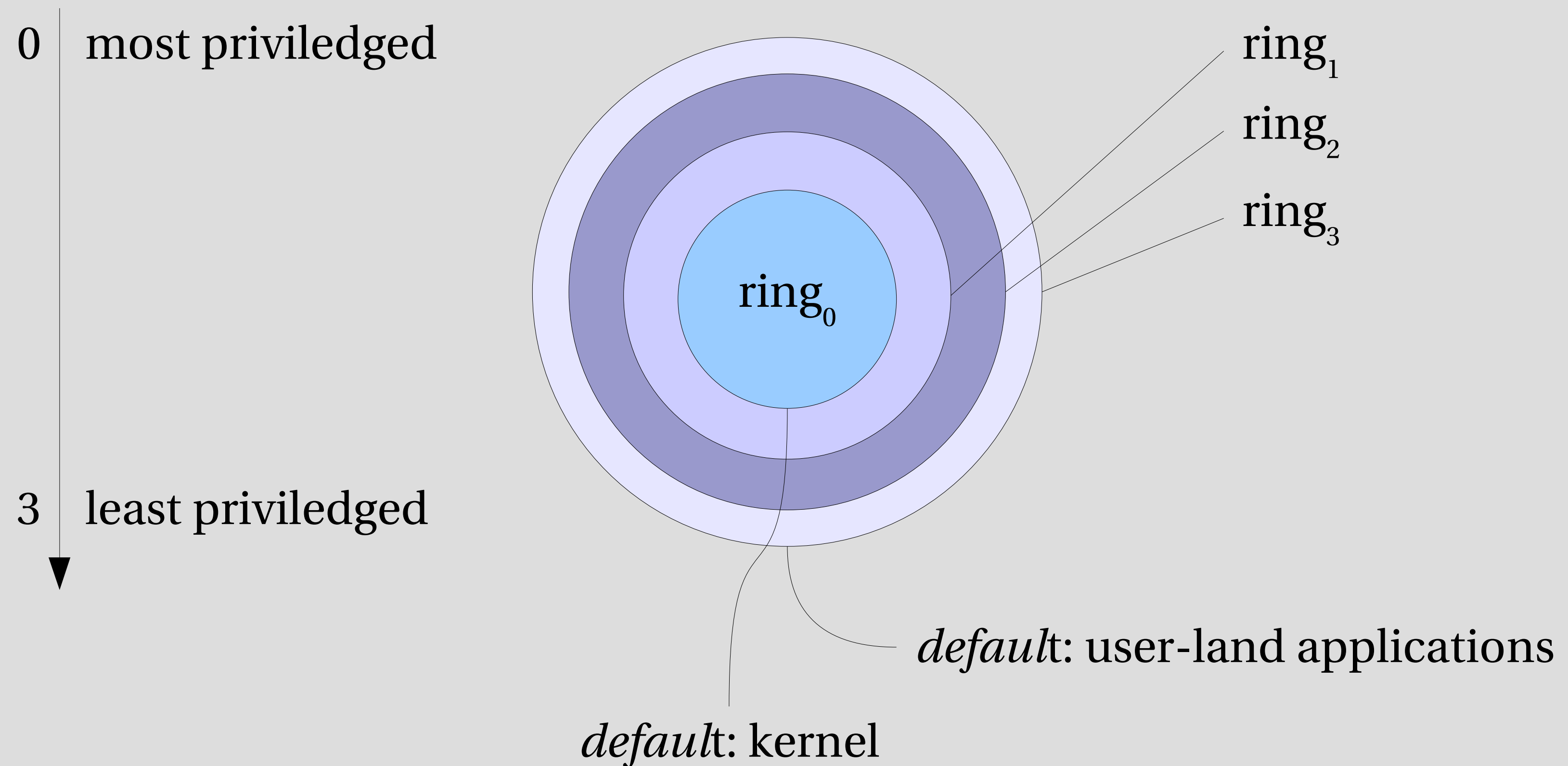
kQEMU implementation

- in order to optimise memory accesses, QEMU also provides kQEMU, an acceleration mechanism implemented in kernel space
- kQEMU only works when the emulated host's CPU is the same as the host CPU

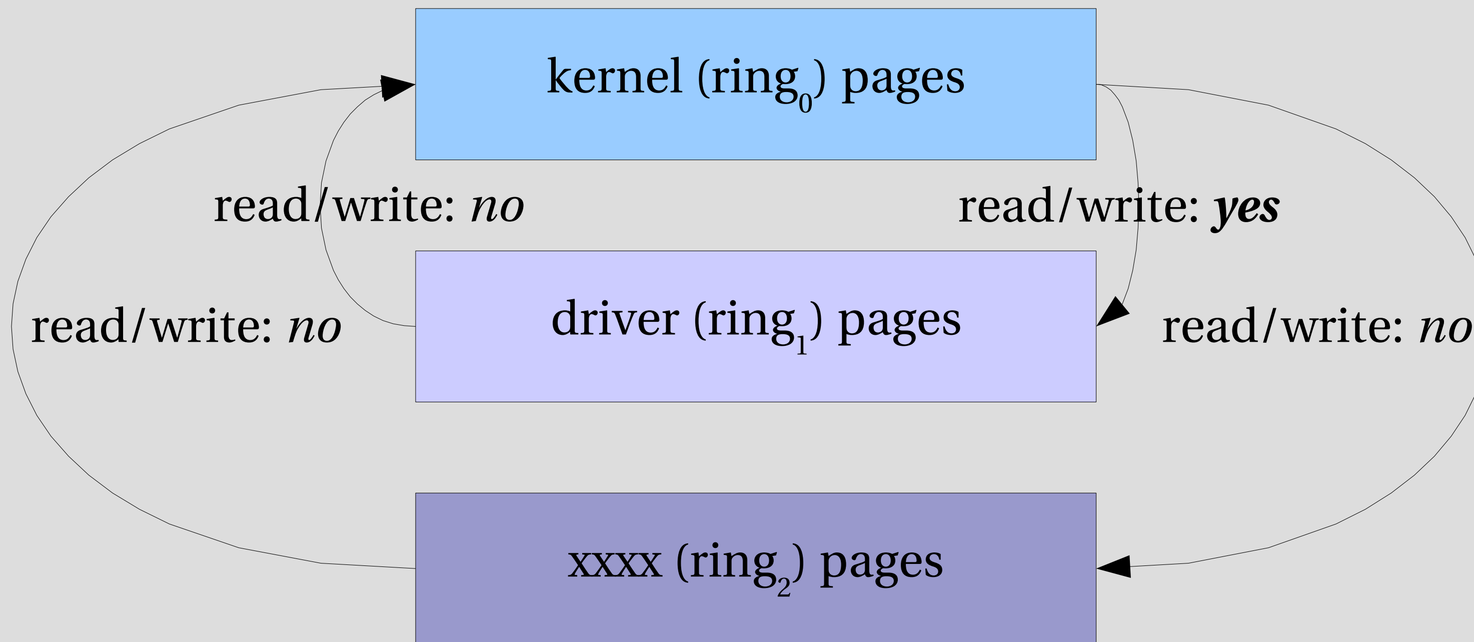
kQEMU implementation

- kQEMU uses some processor features in order to execute faster: CPL
- CPL (current protection level) refers to as "ring protection", a hardware-enforced of task or process data access protection
- this is an x86 mechanism, although most other chips have an equivalent feature

ring protection



ring protection

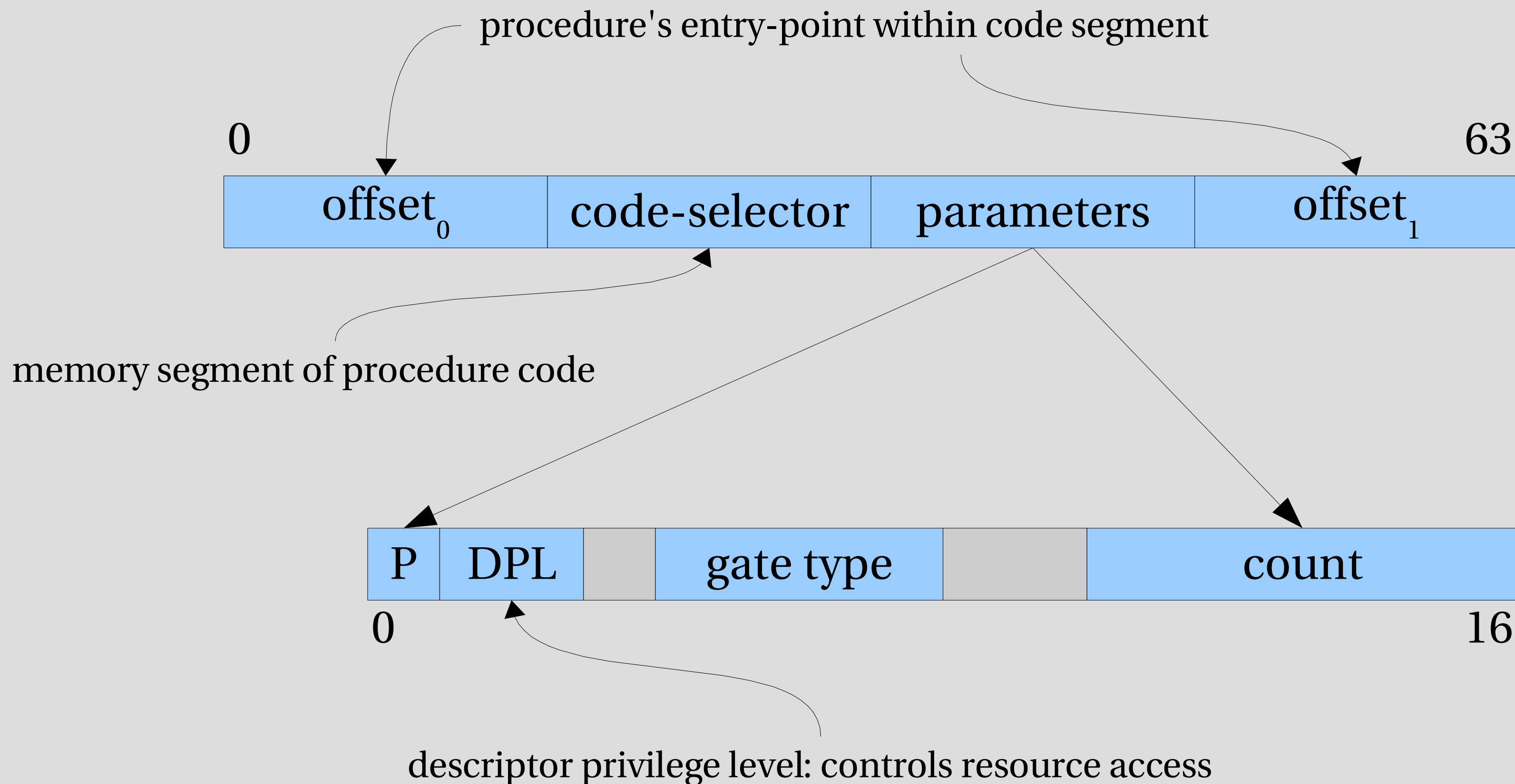


ring protection

- transitions from outer to inner rings occur by a "call gate" functions: data structure in "system" memory normally not accessable for modification
- handling of data-values (parameters) requires special handling
- thus, different stacks provided at each protection level; transition from one protection level to the next accompanied by "stack-switch" operation

ring protection

- call-gate descriptor (call-gate rights)



ring protection

- sample access routine

```
0001    lcall      $callgate-sel, $0
```

0x90 — segment field

- if CPL ≤ DPL, ring-transition occurs
- CPL (current privilege level) based on 2 LSB in CS register

ring protection

1. push SS:SP values onto new stack-segment
2. copy specified parameters onto new stack
3. push CS:IP onto new stack
4. load new CS:IP values onto SS:IP

kQEMU implementation

- kQEMU loads guest images into $ring_3$ and maintains a shadow page table in order to completely segment the guest image from the host operating system
- the "monitor" QEMU interfaces with kQEMU via `/dev/kqemu` device `ioctl(2)`

assisted virtualisation

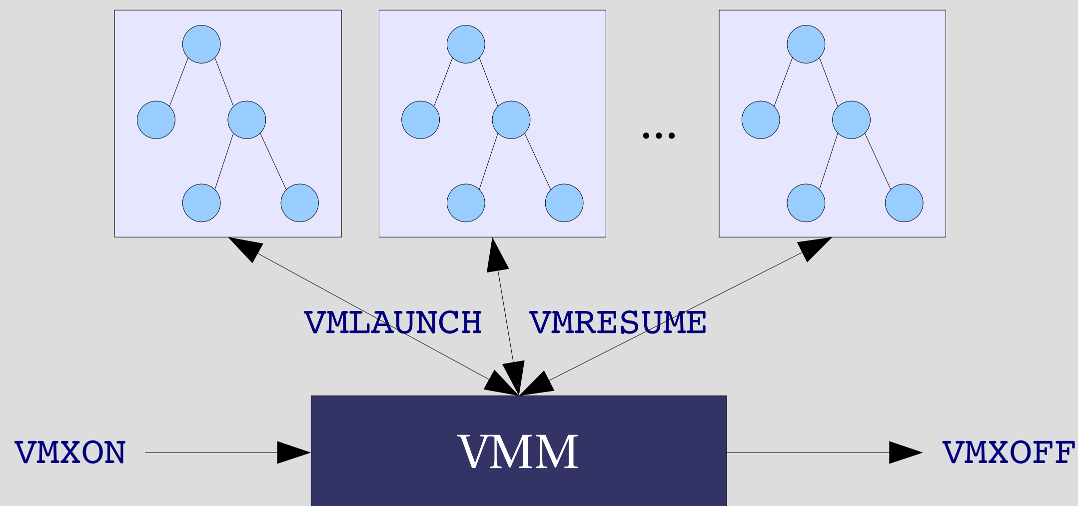
- many other assisting technologies
  - Intel VT ("Vanderpool")
  - AMD-V ("Pacifica")
- typically, hardware virtualisation facilities are very underused, as they're quite complicated to implement
- since ring transitions can be expensive, they still don't provide a high-performance solution to VM trapping

assisted virtualisation

- Intel VT ("Vanderpool")
  - segments instruction execution into VMX-root and VMX-nonroot (beyond "normal" machine operation)
  - VMX-root contains VMM (virtual machine monitor)
  - VMM loads guest VMs into VMX-nonroot
  - VMX-nonroot guests have full hardware facilities (including $ring_0$) at their disposal

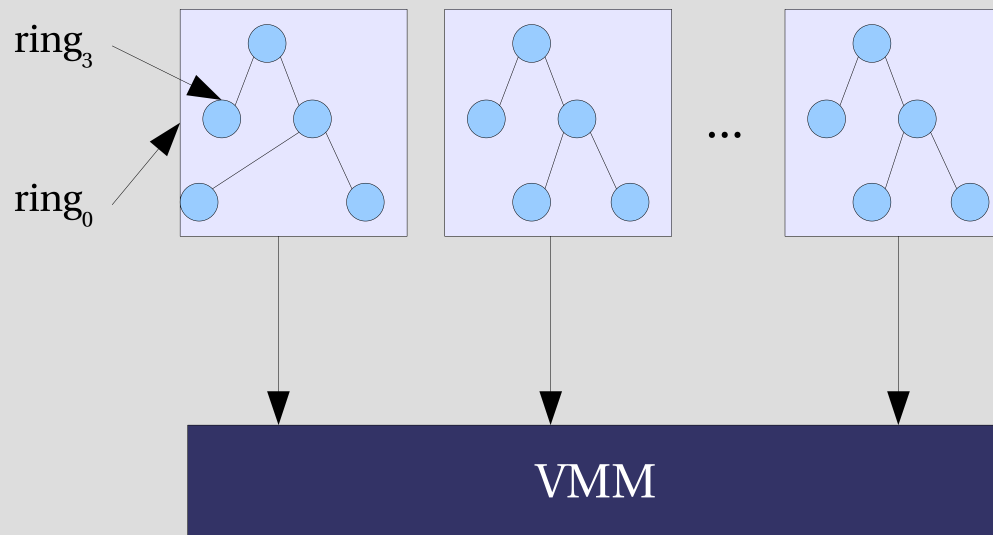
assisted virtualisation

- Intel VT ("Vanderpool")
  - a VMM is created with `VMXON`, at which point VMs may be launched with `VMLAUNCH`



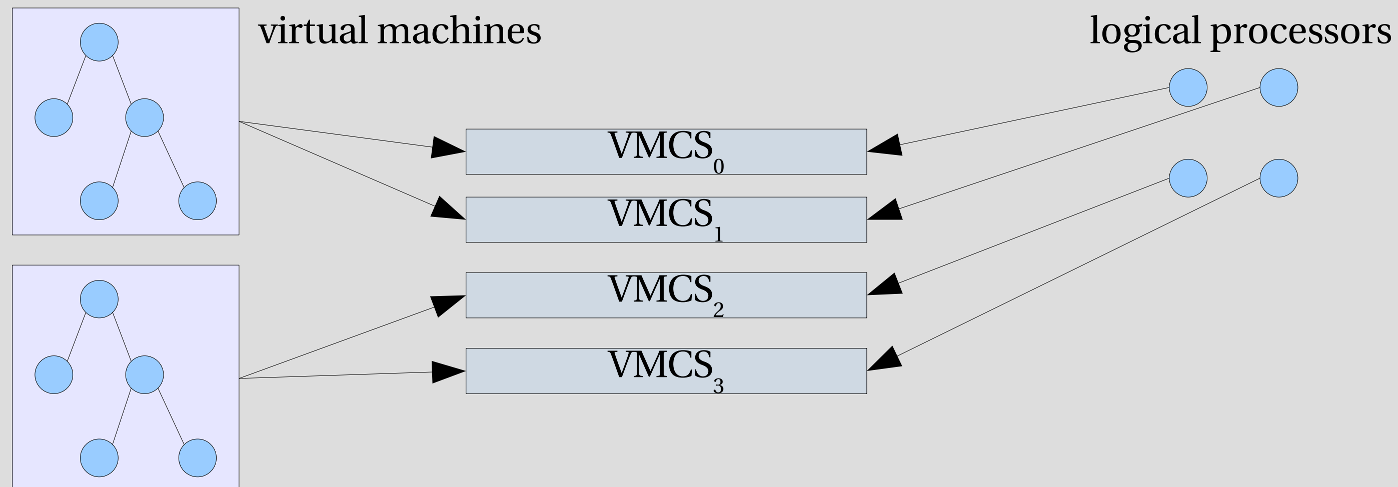
assisted virtualisation

- Intel VT ("Vanderpool")
  - guest operating systems can operate in any ring; the VMX system traps on instructions regardless of their CPL



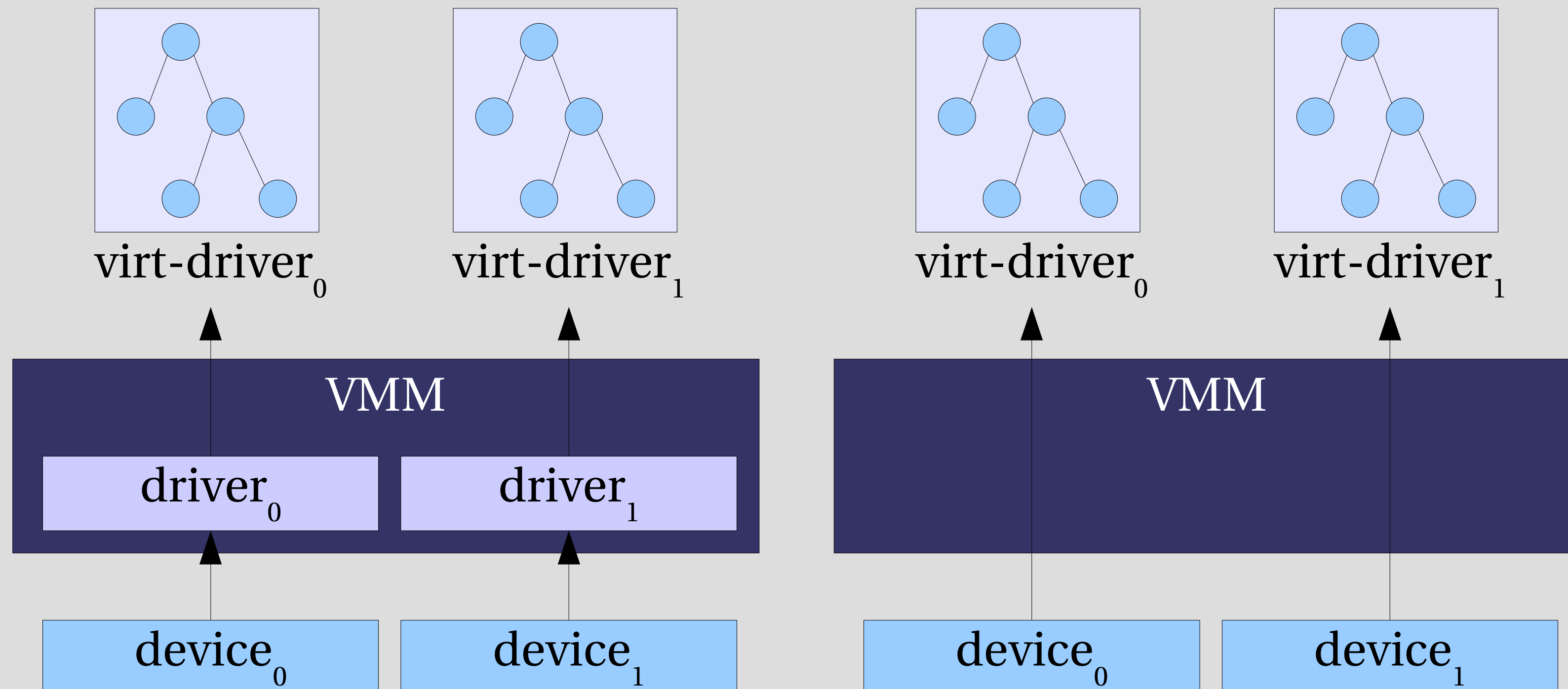
assisted virtualisation

- Intel VT ("Vanderpool")
  - VMCS entities describe logical processors (virtual machine control data structure)
    - VMCS contains argument region, memory limits, trap events, and other VM data



assisted virtualisation

- Intel VT ("Vanderpool")
  - DMA-remapping allows DMA addresses to be directly mapped into "domains"



assisted virtualisation

- Intel VT ("Vanderpool")
  - by mapping I/O domains to devices, the kernel can guarantee complete I/O segmentation between host and guest operating systems
  - this is also useful for *many* other tricks
  - VT also provides interrupt re-mapping and routing, where the VMM may register interrupt sources with particular domains
  - this significantly speeds up I/O interrupt-heavy routines (data availability), which otherwise would need to be demuxed in software

assisted virtualisation

- AMD-V provides an equivalent set of functionality as Intel VT

assisted virtualisation

- cons
  - inherits most of full virtualisations issues, although may be significantly faster (depending on hardware assistance) and simpler (depending on hardware assistance)
  - may be limited to particular architectures (depending on hardware assistance)

assisted virtualisation

- pros
  - inherits most of full virtualisation issues, may also be much faster and simpler (given hardware assistance)

case study

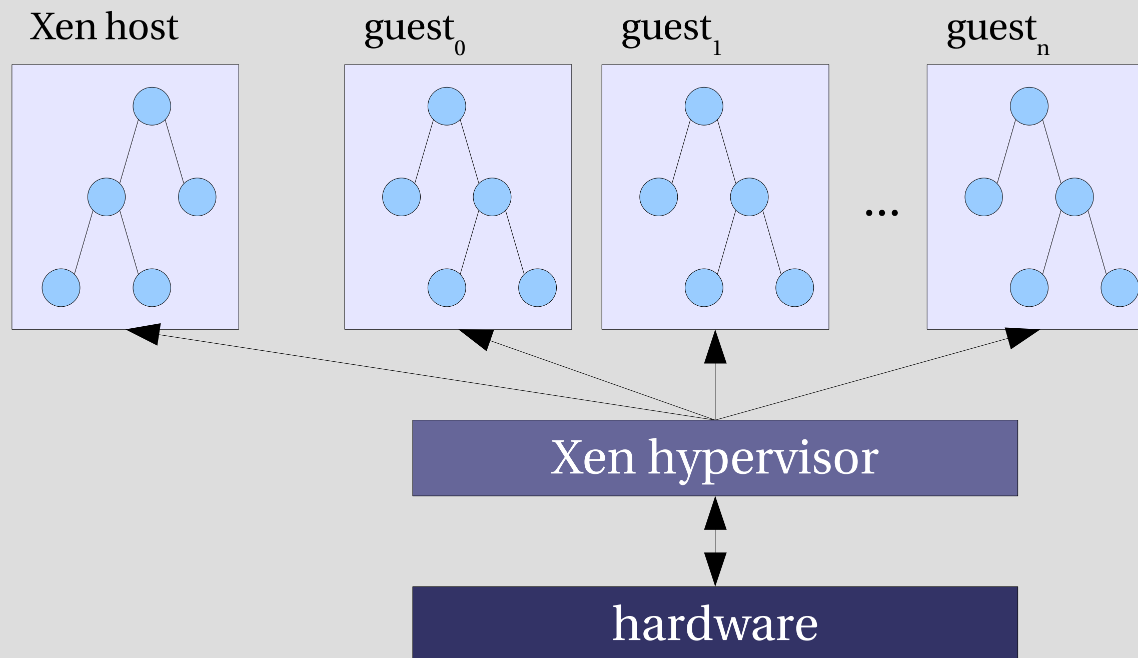
- consider the para-virtualisation provided by Xen (Xen also supports assisted-full virtualisation)
- examples are from NetBSD/Xen 3.1

Xen implementation

- Xen is a complicated mixture of a "para-virtualiser" and an assisted virtualiser
- our analysis focusses on Xen as a true para-virtualiser, or in other words, requiring guest operating systems to be re-written with the Xen "hypercall" ABI as their hardware interface

Xen implementation

- Xen first loads the Xen (hypervisor and control plan software) into $ring_0$, then guest images into $ring_1$ (applications in $ring_3$)



Xen implementation

- the Xen host ("control pane server") is responsible for managing guest domains
- the hypervisor brokers resources to and from guests and host, or in some cases, allows direct access to hardware
- guests (implementing the hypercall ABI) operate as standard operating systems

Xen implementation

- consider the flow of control from hypervisor to and from a guest in the event of a page fault on a guest application

Xen implementation

1. guest application causes page fault
2. Xen forwards page fault via trap table
3. guest finds (e.g.) that no page mapping exists to that page table
4. guest acquires new page
5. guest translates physical to machine address
6. guest notifies MMU via Xen
7. guest requests page-pinning by Xen
8. guest insert page into page table

Xen implementation

- hypervisor notified by CPU fault
  - sys/arch/xen/i386/vector.S

```
646     IDTVEC(trap0e)
647         INTRENTRY
648         movl    TF_TRAPNO(%esp),%eax
649         movl    $T_PAGEFLT,TF_TRAPNO(%esp)
...
654         pushl   %eax
655         movl    %esp,%eax
656         addl    $4,%eax
657         pushl   %eax
658         call    _C_LABEL(trap)
659         addl    $4,%esp
660         addl    $4,%esp
661         testb   $CHK_UPL,TF_CS(%esp)
662         jnz     trap0e_checkast
...
667         jmp     6f
```

Xen implementation

- hypervisor notified by CPU fault
  - sys/arch/xen/i386/vector.S

```
683    6:   STIC(%eax)
684         jz      4f
685         call    _C_LABEL(stipending)
```

Xen implementation

- guest notified asynchronously
  - sys/arch/xen/include/xen.h

```
159    #define __sti()                                              \
160    do {                                                         \
161        volatile shared_info_t *_shared = HYPERVISOR_shared_info;   \
162        __insn_barrier();                                        \
163        _shared->vcpu_data[0].evtchn_upcall_mask = 0;            \
164        x86_lfence(); /* unmask then check (avoid races) */      \
165        if (__predict_false(_shared->vcpu_data[0].evtchn_upcall_pending)) \
166            hypervisor_force_callback();                         \
167    } while (0)
```

Xen implementation

- guest receives page fault notification

```
002	page_fault_handler(virt_addr)
003	{
004		pgdir_entry = virt_to_pgdir(virt_addr);
005		pgtab_entry = virt_to_pgtab(virt_addr);
006		phys_page = virt_to_phys(virt_addr);
```

  - guest acquires new page

```
007		pgtab = getpage();
008		pgtab_mach_addr = mach_to_phys(pgtab);
```

  - guest updates MMU via hypervisor

```
009		HYPERVISOR_MMU_update(MMUEXT_PIN_L2_TABLE, pgtab_mach_addr, ...);
```

Xen implementation

- hypervisor calls: synchronous drops
  - sys/arch/xen/include/hypervisor.h

```
534	static __inline int
535	HYPERVISOR_mmu_update(mmu_update_t *req, int count, int *success_count)
536	{
537		int ret;
538		unsigned long ign1, ign2, ign3;
539	
540		__asm__ __volatile__ (
541			TRAP_INSTR
542			: "=a" (ret), "=b" (ign1), "=c" (ign2), "=d" (ign3)
543			: "0" (__HYPERVISOR_mmu_update), "1" (req), "2" (count),
544				"3" (success_count)
545			: "memory" );
546	
547		return ret;
548	}
```

Xen implementation

- memory management
  - guests must allocate and manage their own physical page tables
  - guest page tables are read only: updates are passed (or batch-passed) to the hypervisor
  - the hypervisor manages actual updates in negotiation with the MMU
  - the hypervisor claims that upper 64 MB of the address space to prevent TLB flush on entering or exiting the hypervisor

Xen implementation

- CPU management
  - guest kernels run at $ring_1$, apps at $ring_3$
  - hypervisor runs at $ring_0$
  - due to ring usage, events must trap into the hypervisor and evaluate local to that CPL
  - page faults and system calls occur with the highest frequency
  - system calls can generally route directly into the guest kernel, without hypervisor intervention

Xen implementation

- I/O management
  - in general, I/O is channeled to and from guests through a set of device abstractions
  - I/O propogates on a ring in the hypervisor
  - Xen emulates interrupts through "events" (like the page fault) passed into guest kernels
  - the ring buffer references I/O descriptors: I/O buffers are maintained out-of-band in guest domain memory
  - Xen may re-order requests in order to optimise for the underlying medium

the end

- byeeeee