Secure Operating System Design and Implementation
Xen

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

March 2, 2012

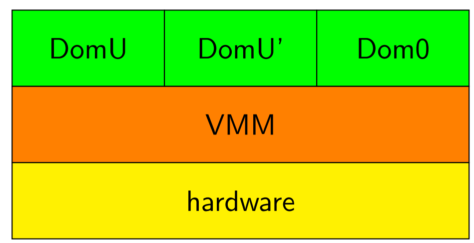
Part I

Xen Overview

Overview

- Xen is a VMM (also called a Hypervisor)
- Xen was originally written to support paravirtualizaiton
- And Linux and Windows were ported to Xen
- But Windows was never more than a proof of concept
- Xen has been extended to support AMD and Intel virtualization extensions
- Xen is the default base for cloud computing

Non-interpreter VMs



- Small VMM, 100K lines of code
- Huge Dom0, 12M lines of code
- Arbitrary number of DomU's
- Each DomU is created with a new number
- Work split between VMM and Dom0

Processor assumptions

- We're going to assume that at most one processor is executing in DomU
- This simplifies concurrency control within the Kernel
- (Without this assumption, the kernel needs to lock all resources before use to prevent race condition, needs to worry about memory consistency)
- Note that this is the default for Xen
- Enabling multiple processors to concurrently execute in the OS is a large undertaking, it took 3 years to do this with Linux

Things to ignore

There are a lot of things that are not relevant for now. Ignore all discussion of

- HVM (hardware virtual machine). (We will run paravirtualized).
- Hypercalls which can be made only from Dom0 .
- which are a way of writing code which is both paravirtualizable and natively executable
- 64-bit code, we are running 32-bit code (even on a 64-bit hypervisor)

Xen expects all 32-bit Intel code to be PAE enabled. This effects only paging.

Xen vs. Architecture

Architecture	**Xen**	**purpose**
rings	rings/regions	privilege and isolation
interrupt	event	asynchronous notification
bios	XenStore start_info	information used to configure OS
system calls	hypercall	invoke more privileged code
virtual memory	virtual memory	memory management
devices	device front-ends	I/O
fence	barriers	ensure memory operations ordered between Dom0 and DomU

Part II

Segmentation

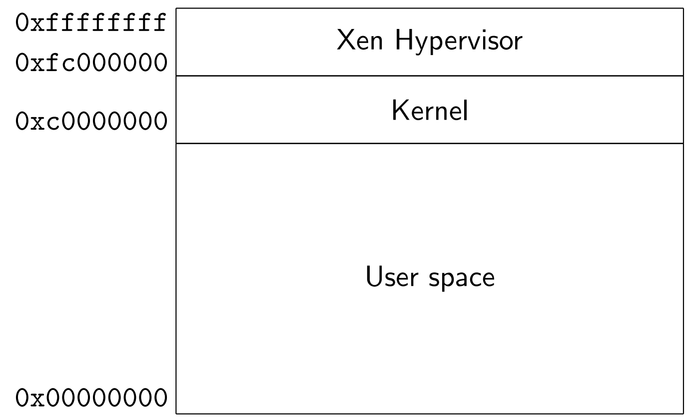
Segmentation

- The x86 architecture supports both
  - Segmentation and
  - Paging
- A virtual address is first translated by segmentation, and then the resulting **linear address** is paged.
- OS designers have largely opted to ignore segmentation, in Linux

```
__KERNEL_CS (Kernel code segment, base=0, limit=4GB, DPL=0)
__KERNEL_DS (Kernel data segment, base=0, limit=4GB, DPL=0)
__USER_CS   (User code segment,   base=0, limit=4GB, DPL=3)
__USER_DS   (User data segment,   base=0, limit=4GB, DPL=3)
```

- Thus Xen starts up with a flat segmented address space
- each segment starts at 0 and is 2^{32} bytes in size.

Memory map



Protection

- Segments
  - CS Code segment: used for instruction accesses
  - SS Stack segment: used for stack accesses
  - DS Data segment: used for data accesses
- In the x86, the CS determines the current ring.
- And privileged instructions can only be executed in ring 0.
- The paging mechanism also determines whether memory can be accessed based on the ring.
- Hence, the major effect of segmentation on OSs is it used to determine the privileged level

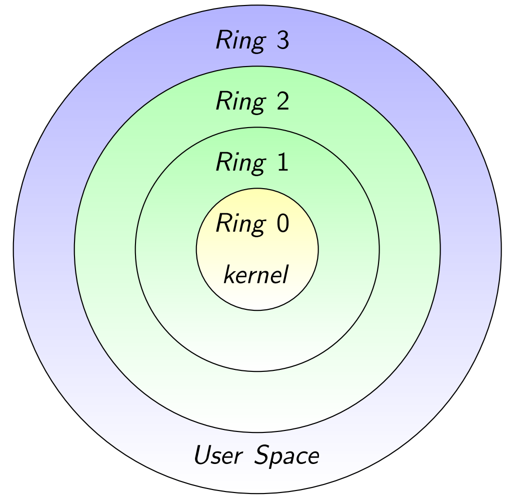
Part III

Rings of Protections

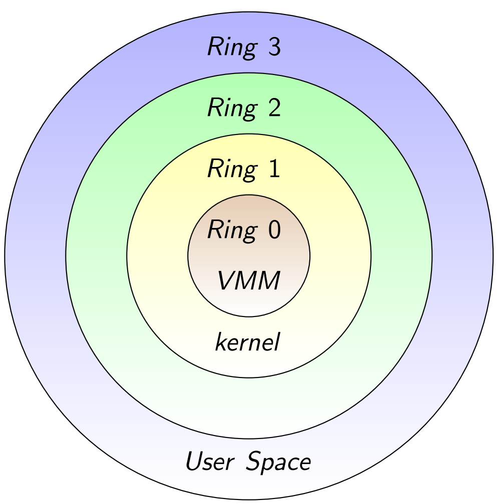
Rings (X86_32)

- x86 architecture has 4 rings of protection
- ring 0 is the most privileged
- normally, an OS runs in ring 0 and user space runs in ring 3
- this is a traditional UNIX model
- but it is possible to use the rings to provide better isolation, for example OS kernel in ring 0 and device drivers in ring 1
- ring i has limited access to ring $j < i$ and unlimited access to $k \geq i$
- Typical setup: Xen runs in ring 0, OS runs in 1, user space in ring 3

Traditional 32-bit OS on bare metal



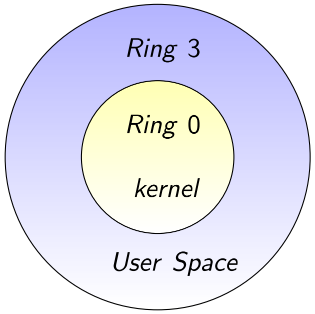
32-bit OS on Xen



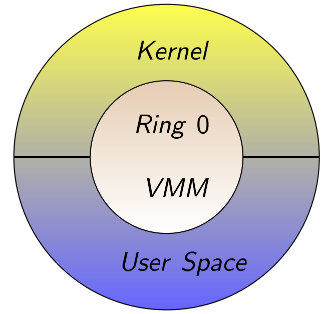
Rings (X86_64)

- For the AMD64, it was decided two levels of privilege suffice
- so it only has ring 0 and ring 3
- Hence, on 64-bit Xen runs in ring 0
- the OS and user space are in ring 3, but separate page tables are used for each.
- Architecture virtualization techniques add ring -1 for a VMM

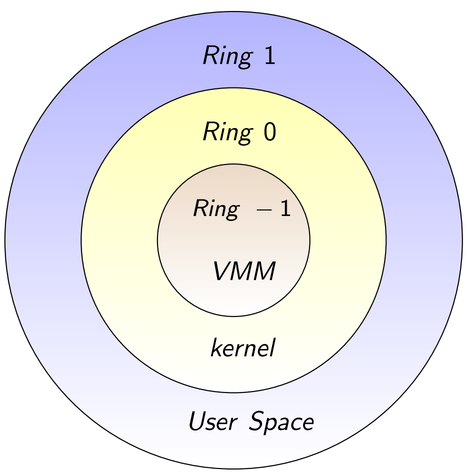
64-bit OS



64-bit OS on Paravirtualized VM



64-bit OS on Hardware VM



Part IV

Hypercalls

Hypercalls

- Only Xen can execute privilege instructions, so the OS must request these from Xen
- To request Xen services, Hypercalls are made
- There are Hypercalls for scheduling, paging, interrupts, ...
- The inline C function invokes a CPP macro, _hypercall2 which takes:
  - A return type
  - The name of the call
  - and 2 arguments, `cmd` and `arg`

```
static inline int
HYPERVISOR_sched_op(int cmd, ulong arg)
  {
    return _hypercall2(int, sched_op, cmd, arg);
  }
```

see include/hypercall-x86_32.h for some calls

Hypercalls (cont'd)

- _hypercall2 (for two arguments) is a C pre-processor macro
- It passes arguments through EAX, EBX, and ECX registers
- It **CALL**s an address, with a (4096 byte) **hypercall page**
- Each call, such as sched_op, uses 32 bytes on the hypercall page
- This means that up to 128 hypercalls are possible, although 45 or so are in use
- _hypercall2 is ugly because
  - It uses advanced macro features
  - It embeds assembly language (`asm volatile`) in C code
  - It is architecture/Xen specific

Hypercall page

- The hypercall page is mapped into the kernel (presumably execute only)
- It contains the code that invokes Xen Hypercalls (e.g., `int 82`)
- and returns the value of the hypercall to _hypercallx
- By supplying this page, Xen controls the code which is on it and thus can migrate to new interfaces if necessary.

Part V

Traps and system calls

System calls

- In addition to Hypercalls (from kernel to Xen), system calls (from user space to kernel) are needed
- Typically made with an interrupt x80 (alt. SYSENTER/SYSEXIT or SYSCALL/SYSRET)
- Parameters passed in registers (typ. EAX, EBX, ECX)
- This causes control to pass to Xen
- Xen then creates a register packet and then invokes the OS

Traps

- ethos/arch/x86/traps.c
- Submits a virtual IDT to the hypervisor.
- This consists of tuples
  - interrupt number
  - privilege ring
  - CS:EIP of handler.
- The 'privilege ring' field specifies the least-privileged ring that
- can trap to that vector using a software-interrupt instruction (INT).
- does not effect hardware induced traps, which go to Xen since they are not necessarily associated with the currently executing Domain.

Xen trap table

```
static trap_info_t trap_table[] = {
  {  0, 0, __KERNEL_CS, (ulong)divide_error },
  {  1, 0, __KERNEL_CS, (ulong)debug         },
  {  3, 3, __KERNEL_CS, (ulong)int3          },
  {  4, 3, __KERNEL_CS, (ulong)overflow      },
  {  5, 3, __KERNEL_CS, (ulong)bounds        },
  {  6, 0, __KERNEL_CS, (ulong)invalid_op    },
  {  7, 0, __KERNEL_CS, (ulong)device_not_available
  {  9, 0, __KERNEL_CS, (ulong)coprocessor_segment_
  { 10, 0, __KERNEL_CS, (ulong)invalid_TSS   },
  { 11, 0, __KERNEL_CS, (ulong)segment_not_present}
```


Xen trap table

```
  { 12, 0, __KERNEL_CS, (ulong)stack_segment},
  { 13, 0, __KERNEL_CS, (ulong)general_protection},
  { 14, 0, __KERNEL_CS, (ulong)page_fault},
  { 15, 0, __KERNEL_CS, (ulong)spurious_interrupt_bu
  { 16, 0, __KERNEL_CS, (ulong)coprocessor_error},
  { 17, 0, __KERNEL_CS, (ulong)alignment_check},
  { 19, 0, __KERNEL_CS, (ulong)simd_coprocessor_err
  {0x80,3, __KERNEL_CS, (ulong)syscall},
  {    0,0,     0,                    0}
};

void
init(void)
{
  HYPERVISOR_set_trap_table(trap_table);
}
```

Part VI

Interrupt use

Interrupts and Xen events

- System calls and Hypervisor calls are transitions from less privileged to more privileged.
- They are caused by the less privileged level asking for privileged services
- But it is also necessary for the more privileged levels to assert control over what can call it and how the call is handled
- At the hardware level, this is done through interrupts
- Which bring control to a (presumably lower) level
- The level is determined by the code segment
- The code segment embeds the privilege (i.e., ring) level
- The Xen to Kernel corresponding are called (Xen) events

Interrupts/events

- Events (Interrupts) cause transitions between levels.
- From the kernel viewpoint,
  - these transitions look like system calls
  - with the exception that they can occur when already in the kernel.
- In terms of the less privilege level, it is not expecting an interrupt
- So after an interrupt, it is important to return to the less privileged level as if no interrupt occurred.
- At the hardware level, this is called **precise interrupts** in which the user-visible state is
  - preserved at the time of interrupt
  - restored upon return from interrupt

Interrupt code in the kernel

- Since an interrupt/event can arrive when the kernel is doing something else
- It is important to prevent race conditions
- It is possible to lock all data structures used by the interrupt
- But locking must be done in both the interrupt code (small) *and* the non-interrupt code
- To simplify this issue, minimum processing is done at the time of interrupt (**upper half**)
- For example, copying packet from an ethernet device
- And the remainder of processing deferred to some convenient time (**bottom half**)
- Such as just before the kernel is going to return to user space

Advantages of split interrupt handling

- Locks are minimized in upper half and in the remainder of the kernel
- Lock bugs are really unpleasant
  - Forget a lock, and you have race condition which is difficult to debug
  - Locks also have performance implications since they block execution
- Upper half does all time-critical functions,
  - maximizing concurrency of CPU with devices
  - preventing lost interrupts and thus lost packets

Part VII

Xen events

Xen event use

- There are a fixed number of interrupt types in a processor
- There is a Xen event **channel** for each of these interrupts
- An endpoint of a channel is called a **port**
- In addition, channels may be created for inter-domain communication
- A Xen event must be sent along a channel
- to send an event an entity in Xen, must have access to the channel.
- To receive an event, must have a handler (procedure) associated with the event in the receiver.
- Obviously, if this is inter-domain communication
  - the channel must be created (receiver)
  - the handler must be installed for that event (receiver)
  - the senders must be informed of the name of the channel

Event types

- Three broad categories of events:
  - **interdomain**,
  - **physical IRQ**, and
  - **virtual IRQ**
- physical IRQ are for Domain 0 or a driver Domain
- virtual IRQ are for virtualized devices, such as clock, console
- interdomain events are used for data exchange between domains (and are used to indicate data waiting or consumed.
- interdomain events, unlike interrupts, are bidirectional

Binding the timer virtual interrupt

```
evtchn_bind_virq_t op;

op.virq = VIRQ_TIMER;
op.vcpu = 0;
int status = HYPERVISOR_event_channel_op(
     EVTCHNOP_bind_virq, &op);
if (0 != status) { /* handle error */ }
```

- The hypercall set the port number, `op.port`.
- Some channels are bound at Domain load time, including
- XenStore and console

Allocate an unbound interdomain event channel

```
evtchn_alloc_unbound_t op;
op.dom = DOMID_SELF;
op.remote_dom = remote_domain;
int status = HYPERVISOR_event_channel_op(
            EVTCHNOP_alloc_unbound, &op);
if (0 != status) { /* handle error */ }
```

- An interdomain event channel has 2 domains, the *creating* and *opposite* domains.
- The creating domain creates the channel (and its associated port)
- op.port contains port number

Remote binding of an interdomain event channel

- The creating domain sends to the opposite domain creating_port and creating_domain
- These values are typically sent by writing to the XenStore
- The opposite domain binds to the event channel

```
evtchn_bind_interdomain_t op;
op.remote_port = creating_port;
op.remote_dom = creating_domain;
int status = HYPERVISOR_event_channel_op(
        EVTCHNOP_bind_interdomain, &op);
if (0 != status) { /* handle error */ }
```


Getting started

- Its a good idea to start with all the events masked
- typically, OS initialization brings up services one at a time and it is only after a service is initialized that the OS is ready to handle the corresponding event.
- when binding a new event, shared_info.evtchn_pending[0] bit should be cleared
- event delivery is disabled at boot time. So it is necessary to
  - clear shared_info.vcpu_info[0].evtchn_upcall_mask
  - check shared_info.vcpu_info[0].evtchn_upcall_pending and if set handle the event and clear the bit.
- Note that 0 is for VCPU 0, the only CPU in use by nanoOS.

Sending an (interdomain) event

```
struct evtchn_send event;
event.port = portNumber;
int status = HYPERVISOR_event_channel_op(
                 EVTCHNOP_send, &event);
if (0 != status) { /* handle error */ }
```


Initializing events

```
static evtchn_handler_t handlers[NUM_CHANNELS];

void EVT_IGN(evtchn_port_t port, struct pt_regs * regs) {};

// Initialise the event handlers
void init_events(void)
{
  // Set the event delivery callbacks
  HYPERVISOR_set_callbacks(
      FLAT_KERNEL_CS, (ulong) hypervisor_callback,
      FLAT_KERNEL_CS, (ulong) failsafe_callback);
  // Set all handlers to ignore, and mask them
  for (uint i=0 ; i<NUM_CHANNELS ; i++)
  {
    handlers[i] = EVT_IGN;
    SET_BIT(i, shared_info.evtchn_mask[0]);
  }
  // Allow upcalls.
  shared_info.vcpu_info[0].evtchn_upcall_mask = 0;
}
```


find a vcpu which can process the event on the channel

This describes what Xen does to find a VCPU for an event

```
findEligibleVCPU(channel)
if channel bound to vcpu then
   if unmaskedEvents(vcpu) then
      return {vcpu}
   end if
else
   return {v : v ∈ vcpuSet|unmaskedEvents(v))}
end if
return {}
```


Event masking

This describe what happens (In Xen) when an event arrives

```
if event already pending on channel then
   return
end if
set pending bit for channel
if channel masked then
   return
end if
if ∃ vcpu ∈ findEligibleVCPU(channel) then
   set vcpu's pending flag
   set vcpu's event selector
   deliver event via upcall
end if
```

Callbacks

- The number of events per domain is limited to 1024 (32 bits per word times 32 words)
- A default handler, EVT_IGN which when called does nothing
- like all handler, it is passed the port number and the registers
- HYPERVISOR_set_callbacks gives two callback (from Xen to DomU)
- hypervisor_callback calls the specific handler for the event, it is assembly language which calls C
- for each possible channel, the default handler is defined and the corresponding evtchn_mask is set (still not accepting events)
- Now the CPU's events are unmasked

Event callbacks

```
uint eventPendingAndUnmasked(uint word) {
  return shared_info.evtchn_pending[word] &
         ~shared_info.evtchn_mask[word];
}

void do_hypervisor_callback(struct pt_regs *regs) {
  vcpu_info_t *vcpu = &shared_info.vcpu_info[0];
  // Make sure we don't lose the edge on new events
  vcpu->evtchn_upcall_pending = 0;
  // Set the pending selector to 0 and
  // get the old value atomically
  uint pendingSelector
        = xchg(&vcpu->evtchn_pending_sel, 0);
```


Event callbacks

```
while(pendingSelector != 0)
{
  // Get the first bit of the selector and clear it
  uint eventWord = first_bit(pendingSelector);
  pendingSelector &= ~(1 << eventWord);
  uint event;

  // While events are pending on unmasked ports (book bug)
  while(event = eventPendingAndUnmasked(eventWord))
  {
    // Find the first waiting event in the eventWord
    uint eventBit = first_bit(event);

    // Combine the two offsets to get the port
    evtchn_port_t port = (eventWord << 5) + eventBit;
    // Handle the event
    handlers[port](port, regs);
    // Clear the pending flag
    CLEAR_BIT(shared_info.evtchn_pending[0], eventBit);
  }
```


Other issues with event call backs

- hypervisor_callback is an assembly language routine
- it is invoked by the Xen hypervisor when both the event and VCPU is unmasked
- It calls the C routine do_hypervisor_callback
- Which calls the individual handlers
- Still need to set up real handlers
- these are set up one at a time as associated service is initialized and then
- The event channel is unmasked

Use of event handlers

- Initialize everything, including handlers
- Then enter the main event loop
- When a hardware event occurs, the interrupt mechanism disables interrupts
- Hardware interrupts enabled by return from interrupt, Xen events enabled by assembly language code for rti processing

```
// main event loop
while (1) {
   __cli();  // disable events
   // do all periodic processing, set time events
   bottomhalf();
  __sti();  // re-enable events

  // block OS until a new event occurs
  HYPERVISOR_sched_op(SCHED_OP_block, &op);
}
```


Event summary

- pseudo device event channels must be bound to by DomU (except for the prebound devices)
- Interdomain event channels must be created in one domain and bound to in a different domain.
- An event can be sent with a hypercall on the channel's port
- An event can be received by polling, but a more typical mechanism is to use a callback—a procedure which is invoked when an unmasked event arrives.
- Need to inform Xen of the callbacks
- One callback needs to invoke callbacks for each individual event

Part VIII

System start

System Start

- When an OS starts, it interrogates its surroundings (BIOS, etc.) to determine the system configuration.
- Thus it can determine whether it can run on the hardware, and how to organize itself.
- In Xen, this mechanism is replaced with the start_info_page
- This page includes information which does not change (unless the VM is migrated–i.e. resumed)
- The start info page points shared_info
- shared_info page is shared by Xen and the domain
- it contains information which changes during execution
- shared_info includes interrupt/events and time keeping

Start info

```
struct start_info {
// The following are filled in both on initial boot and on re
char magic[32];        // "Xen-v.s".  where v is version and s s
ulong nr_pages;        // Total pages allocated to this domain.
ulong shared_info;     // MACHINE address of shared info struct.
uint32 flags;          // SIF_xxx flags.
xen_pfn_t store_mfn;  // MACHINE page number of shared page fo
uint32 store_evtchn;  // Event channel for XenStore
union {
  struct {
    xen_pfn_t mfn;   // MACHINE page number of console page.
    uint32  evtchn; // Event channel for console page.
  } domU; //  (dom0 console info not included)
} console;
// The following are filled on initial boot, but not resume.
ulong pt_base;        // VIRTUAL address of page directory.
ulong nr_pt_frames; // Number of bootstrap p.t. frames.
ulong mfn_list;       // VIRTUAL address of page-frame list.
...
int8_t cmd_line[MAX_GUEST_CMDLINE];
```


Start info notes

- in general, nr_pt_frames $\leq$ nr_pages
- The difference is unallocated pages which can be mapped and used for
  - OS data structures
  - User space pages
- mfn_list pseudo-physical to machine frame number map, i.e. ((ulong$*$) mfn_list)[p] is the machine frame number associated with physical frame number p.

Shared info

This data structure is stored in a page which is shared between Xen and the domain

```
struct shared_info {
  struct vcpu_info vcpu_info[MAX_VIRT_CPUS];
  ulong evtchn_pending[sizeof(ulong) * 8];
  ulong evtchn_mask[sizeof(ulong) * 8];
  // Wallclock time: updated only by Xen.  Guest's
  // gettimeofday() syscall based on wc values.
  uint32 wc_version;// Version counter
                    // (see vcpu_time_info_t)
  uint32 wc_sec;     // Secs  00:00:00 UTC, Jan 1, 1970.
  uint32 wc_nsec;    // Nanoseconds since wc_sec
  struct arch_shared_info arch; // arch specific info
};
```


Shared info comments

The shared info page contains info about

- for each virtual CPU (there can be as many as MAX_VIRT_CPUS although only 1 is initially active.
- There are 1024 possible ports (on 32-bit). book bug (32/word and 32 words)
- The wall clock time is on a shared page, so that Xen maybe updating it while the OS is reading it, so
  - Xen makes the version number odd updating it
  - Xen then updates the value
  - Xen then increments the version number
- Thus, the OS reads the version number, read the values, and re-reads the version number to ensure that the number is both even and unchanged

Virtual CPU Information

Part of the shared_info

```
struct vcpu_info {
 uint8_t evtchn_upcall_pending;
 uint8_t evtchn_upcall_mask;
 ulong evtchn_pending_sel;
 struct arch_vcpu_info arch;
 struct vcpu_time_info time; // CPU time
}; /* 64 bytes (x86) */
```

- evtchn_upcall_pending: set to non-zero by Xen to indicate pending events, cleared by OS. Only set if upcalls are masked.
- evtchn_upcall_mask: if non-zero, no upcall activation. Cleared when VCPU requests a block.
- evtchn_pending_sel: a bit mask where bit i is set if there is a pending event in the ith word, that is port $32i \ldots 32i + 31$.

vcpu time info

```
struct vcpu_time_info {
  // see xen.h for comments
  uint32_t version;
  // the next two values are as of the last time
  // Xen updated them.  To get current time,
  // must do a RDTSC, etc.
  uint64_t tsc_timestamp;   // TSC
  uint64_t system_time;     // Time, in nanosecs,
                            //  since boot.

  uint32_t tsc_to_system_mul;
  int8_t   tsc_shift;
}
```


Part IX

Kernel startup code

start_kernel

```
char stack[8192]; // stack used by start_kernel

// shared_info declared in assembly, typed here
extern volatile shared_info_t shared_info; // volat

void start_kernel(start_info_t * start_info)
{
// Map the shared info page
HYPERVISOR_update_va_mapping((ulong) &shared_info,
    __pte(start_info->shared_info | 3),
    UVMF_INVLPG);

init_events();

// Initialise the console
console_init(start_info);
```

start_kernel (cont'd)

```
// Write a message to check that it worked
console_write("Hello world!\n\r");
console_write("Xen magic string: ");
console_write(start_info->magic);
console_write("\n\r");

// Set up the XenStore driver
xenstore_init(start_info);
// Test the store
xenstore_test();
// Flush the console buffer
console_flush();
// Exit, since we don't know how to do anything els
}
```


Notes

- stack is 8K, storage used in main and set up in the assembly language bootstrap
- updates to virtual address mapping are done by hypervisor calls to ensure that VM's access only their own pages.
- the upper 20 bits of the page table entry (pte) are used for the mapping (substituting page number for frame number).
- the lower 12 bits of the pte are used for flags
- The flags set must be 3, the low order bit to indicate that the page is present and the next bit to enable read and write privileges.
- UVMF_INVPG ensures the TLB is updated

Assembly language (bootstrap)

```
// loader file information for Xen
 section __xen_guest
  .ascii    "GUEST_OS=Nano-OS"
  .ascii    ",XEN_VER=xen-3.0"
  .ascii    ",VIRT_BASE=0x0"
  .ascii    ",ELF_PADDR_OFFSET=0x0"
  .ascii    ",HYPERCALL_PAGE=0x2"
  .ascii    ",PAE=yes"
  .ascii    ",LOADER=generic"
  .byte     0
.text

// declare the globals in this file
.globl _start, shared_info, hypercall_page
```


Assembly language (bootstrap cont'd)

```
// Initial entry point
_start:
 cld                        // Clear direction flag
 lss stack_start, %esp      // Load the stack segment
 push %esi                  // Setup start_info for start_
 call start_kernel          // Jump into the kernel

// Initial stack space.  lss (above) loads initial
// + the stack segment selector (48 bits in all)
// stack segment selector is 13+1+2 bits for
// segment selector number + table + request priv.
stack_start:
      .long stack+8192, FLAT_KERNEL_SS
```

Assembly language (bootstrap cont'd)

```
  // Unpleasant -- the PTE that maps this page is
  // actually overwritten to map the real shared-in
  .org 0x1000
shared_info:

  .org 0x2000
hypercall_page:

  .org 0x3000
```


Bootstrap assembly language notes

- _start called by Xen and specified in loader script as entry point
- Assembly program is in the first 3 pages of memory
- First 4 lines set up stack (memory for stack is in the C routine)
- It pushes the start_info address on the stack (start_kernel 's parameter) and the calls start_kernel
- start_stack describes the 48 bits of stack address and segment descriptor needed to specify stack address
- shared_info is a name declared at virtual address 0x1000. This must be mapped into nanoOS.
- hypercall_page defined and mapped in at virtual address 0x2000
- C code begins at address 0x3000

Part X

Time

Time

- Two types of clocks, a cycle counter and a wall clock time
- Xen provides these to the OS via shared_info
- Problem: at nano-second granularity, a CPU would need to be dedicated to updating the wall clock time
- Instead, the Time Stamp Counter (TSC) is a hardware counter which can be read when necessary, converted to nano-seconds, and then used to produce a fine-grained wall clock.
- To do this, compute deltaTsc which is the current TSC less the TSC at the time the wall clock time was last updated
- Then scale it
- Then add it to the wall clock time

Uses of time

- Alarms need to be set to ensure *finite progress*
- Application programs often need time
- Time of day is often used to schedule periodic tasks (e.g., cron)
- Files are marked with the time created
- Time is needed to schedule time between processes
- One of the big difference between VMs and bare hardware is time
  - On VMs, time is not cycles
  - So OSs for VMs track virtual time, time which elapses only when the VM is running.

Time

- Need to bind the timer virtual IRQ
- Can then set a timer

```
HYPERVISOR_set_timer_op(uint64 timeout)
```

  using the number of nanosecond from system (domain) boot to determine when the timer goes off.
- need to have a gettimeofday() routine to determine the current time
- Time of day is relative to an **epoch**, Jan, 1, 1970 which is UNIX's birthday.
- So different uses of time are off different bases

Oddities

- shared info pages are mapped into both Xen and the domain
- hence, it may be updated concurrently when read
- so a spin lock is built
- in Xen, updates are signalled using version number

```
version++;    // make version odd
update values
version++;    // make version even
```

- Need to ensure that value not being updated when read and
- Version has not changed after going even until values read

gettimeofday

```
inline uint64
NANOSECONDS(uint64 tsc)
{
 struct vcpu_time_info * time
       = &(shared_info.cpu_info[0].time);
  uint64 scaledTsc =  tsc << time->tsc_shift;
 return scaledTsc * time->tsc_to_system_mul
}

inline bool
isOdd(uint32 v)
{
 return v & 1;
}

#define RDTSC(x) asm volatile ("RDTSC":"=A"(tsc))
```

gettimeofday (start of procedure)

```
// removed unused timezone in the book's version
int gettimeofday(struct timeval *tp)
{
 uint64_t tsc, old_tsc, system_time;
 // Get the time values from the shared info page
 uint32_t version, wc_version;
 uint32_t seconds, nanoseconds;
 struct vcpu_time_info * time
      = &(shared_info.cpu_info[0].time);
```


gettimeofday (reading shared info)

```
do  // Loop until shared info values can be read
{
  do  // Spin if the time value is being updated
  {
    wc_version = shared_info.wc_version;
    version = time->version;
  } while(isOdd(version) || isOdd(wc_version));
  // Read the values
  seconds = shared_info.wc_sec;
  nanoseconds = shared_info.wc_nsec;
  system_time = time->system_time;
  old_tsc = time->tsc_timestamp;
} while(version != time->version
           ||
        wc_version != shared_info.wc_version
        );
```


gettimeofday (updating wc time based on TSC)

```
  RDTSC(tsc); // Get current TSC value
  uint64 deltaTsc = tsc-old_tsc; // cycles since wc
  // Update the system time
  uint64 sinceUpdate = NANOSECONDS(deltaTsc);
  system_time += sinceUpdate
  // Move complete seconds to the second counter
  sinceUpdate += nanoseconds;
  seconds += sinceUpdate / 1000000000;
  nanoseconds = sinceUpdate % 1000000000;
  // Return second & microsecond values (Book Bug?)
  tp->tv_sec = seconds;
  tp->tv_usec = nanoseconds / 1000;

  return 0;
}
```


Time summary

- Time is an important service
- Elapse time for scheduling
- Elapse time is called system time
- Time of day is important for dealing with external entities
- For example, humans or other systems
- CPU power management complicates time calculations
- And other issues effect time scaling as well
- In addition to this, ntp (network time protocol) is used to keep clocks accurate.

Part XI

Console

Console

- A console pseudo device is implemented using events and shared memory
- The machine address for the shared memory page and event for the console is passed in start_info
- Must map the page into virtual memory
- Use events to send notification to Dom0 of NanoOS console output
- Use events to receive notification from Dom0 of NanoOS console input

Console

- A console is the simplest device
- It uses a 2048-byte output buffer
- It also uses a 1024-byte input buffer
- These buffers of circular producer-consumer buffers
- A few variables in_cons, in_prod, out_cons, out_prod
- and signals when data is waiting in the output buffer.
- This same style is used in the XenStore, the block driver, and the network driver
- Goal: get console working as early as possible in the design of an OS so you can have better visibility into the execution

Xen console interface

```
struct xencons_interface {
    char in[1024];
    char out[2048];
    XENCONS_RING_IDX in_cons, in_prod;
    XENCONS_RING_IDX out_cons, out_prod;
}
```

- This data structure **must** be first mapped into a page shared by DomU and Dom0
- There is an event channel allocated for the console at system start

XENCONS_RING_IDX

- the XENCONS_RING_IDX is an **unsigned int**
- the prod side is only incremented by the sender
- the cons side is only incremented by the receiver
- the number of bytes in the buffer are $prod - cons$
- of course this is limited to the size of the buffer
- to find the index i in the buffer b,
  i = MASK_XENCONS_IDX(prod, b)
- which will be between $0 \ldots sizeof(b) - 1$
- Note because the buffer is circular, cannot use memcpy (which assumes a linear buffer)

Free running counters

- Xen's ring buffer use free running counters
- After putting in 2^{32} bytes counter will wrap around
- That is, the XENCONS_RING_IDX value after $2^{32} - 1$ is 0
- Nevertheless, $prod - cons$ always gives the number of bytes in the buffer
- Even if, say $prod = 1$ and $cons = 2^{32} - 5$
- Note, this depends on using 2's complement arithmetic
- See book for more details

Mapping Xen console

- x86 page size is 4096 bytes, thus an address which points to start of page has low-order 12 bits equal to zero
- Rather than a machine address, Xen provides a machine frame number (mfn), which truncates the low-order 12-bits
- First map mfn to a pseudo-physical frame number (pfn)
- Second, compute the pseudo-physical address $p = pfn << 12$
- Third, from physical address p create virtual address p+&_text
- (Xen maps physical addresses to virtual address in order)
- _text is the first virtual address of the kernel (it is given in the loader script)

Mapping the Xen console

```
int console_init(start_info_t * start)
{
  console = (struct xencons_interface*)
    ((machine_to_phys_mapping[
              start->console.domU.mfn] << 12)
             +
     ((ulong)&_text));
  console_evt = start->console.domU.evtchn;
  /* TODO: Set up the event channel */
  return 0;
}
```

Console write

```
// write null terminated string
int console_write(char * message) {
 struct evtchn_send event;
 event.port = console_evt;
 int length = 0;
 while(*message != '\0')
 {
   // 1. wait until space for a char is available
   // 2. write the next char
 }
 HYPERVISOR_event_channel_op(EVTCHNOP_send,
                &event);
 return length;
}
```


Console write (wait for space)

```
// 1. wait until space is available
//    back end consumes bytes in the buffer
XENCONS_RING_IDX data; // ring index
do {
 data = console->out_prod - console->out_cons;
 HYPERVISOR_event_channel_op(EVTCHNOP_send,
                  &event);
 mb();
} while (data >= sizeof(console->out));
```


Console write (write the data)

```
// write the next char
int ring_index = MASK_XENCONS_IDX(
                    console->out_prod,
                    console->out);
console->out[ring_index] = *message;
// Ensure that the data is visible to
// other processors before continuing
wmb();
// Increment input and output pointers
console->out_prod++;
length++;
message++;
```


Console read

- To read the console you'll need an event handler for the console event
- The handler is invoked when there is either
  - waiting input or
  - when bytes are remove by the consumer
- The handler thus can
  - read the input
  - write more output
- We don't want the kernel to ever block unless there is *nothing* to do

Part XII

Virtual Memory

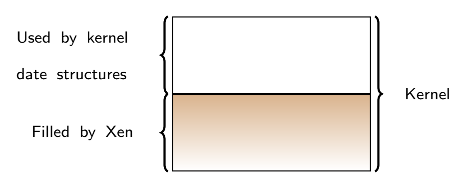
Virtual Memory

- Page tables map **virtual memory addresses** to **physical memory addresses**
- Virtual memory addresses can be distributed through the virtual address space
- For example, their may be a gap between stack and heap to allow for growth of the stack
- But physical addresses are consecutive, with no gaps
- An OS may depend on this property
- When it builds its page tables, which map virtual addresses to physical addresses

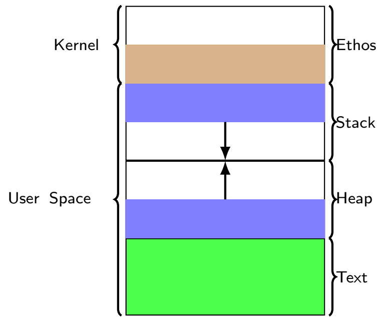
Xen Virtual Memory

- In Xen, physical addresses are allocated to Xen and to the various Doms
- Because of dynamic usage patterns and fragmentation the underlying addresses allocated to an OS may not be contiguous
- Xen's solution
  - machine addresses correspond to the hardware physical address
  - physical addresses are contiguous for a domain
  - virtual addresses are allocated with holes
- Mappings change only the **frame number** (the high order 20 bits), not the address within a page (low order 12 bits)
- Thus, often concerned with mfn, pfn,

Kernel layout in Xen



Process layout in Xen



pfn to mfn

- As part of the shared info pages, Xen provides
  - max_pfn: the maximum page frame number
  - pfn_to_mfn_frame_list
- Xen starts DomUs in virtual real mode, meaning that it sets up the initial page table for the OS
- The mapping of parts of the OS to the memory is controlled by the (OS specific) loader script
- The OS must then load in the Xen specific components (grant pages, start_info , shared info)
- An OS uses only a single page table until it creates its first processes.

Start of day memory layout

1. The domain is started within contiguous virtual-memory region.
2. The contiguous region ends on an aligned 4MB boundary (in Mini-OS it ends at 4MB).
3. Bootstrap elements are packed together, but each is 4kB-aligned.
4. The list of page frames forms a contiguous 'pseudo-physical' memory layout for the domain. In particular, the bootstrap virtual-memory region is a 1:1 mapping to the first section of the pseudo-physical map.
5. All bootstrap elements are mapped read-writable for the guest OS. The only exception is the bootstrap page table, which is mapped read-only.
6. There is guaranteed to be at least 512kB padding after the final bootstrap element. If necessary, the bootstrap virtual region is extended by an extra 4MB to ensure this.

Order of bootstrap elements

1. relocated kernel image
2. initial ram disk [mod_start, mod_len]
3. (The initial ram disk may be omitted.)
4. list of allocated page frames [mfn_list , nr_pages]
5. start_info_t structure [register ESI (x86)]
6. bootstrap page tables [pt_base, CR3 (x86)]
7. bootstrap stack [register ESP (x86)]

Part XIII

Grant Tables

Grant tables

- Events are good for providing a notification, but not for passing lots of data
- To pass information, Xen uses shared memory
- These pages can be shared between domains or
- Between a domain and Xen
- (Xen controls all page tables, so it can enable this sharing)
- The Xen mechanism for this is called **grant tables**
- Grant tables can be used to *transfer* pages between domains,
- but this requires large transfers (many pages) to be efficient.
- Grant tables can be used to *share* pages between domains

Use of Grant Tables

- One or more shared pages are allocated to communicating for each device
- DomU treats this collection of shared pages as a ring buffer (a producer-consumer structure).
- Information is put in/taken out of the ring in FIFO order
- Events inform DomU when there is data waiting
- There are three fundamental devices for Xen

device	buffer contents
Console	printable text
Disk	disk blocks
Network	ethernet frames

Grant entry structure

```
// Xen-3.2,  available for backward compatibility
struct grant_entry_t {
  // GTF_permit_access:
  //        Frame that @domid is allowed
  //        to map and access. [GST]
  // GTF_accept_transfer:
  //         Frame whose ownership
  //         transferred by @domid. [XEN]
  uint16_t flags;

  domid_t  domid;   // domain to share with

  uint32    frame;   // machine frame number
};
```

Informing Xen of shared paged

```
gnttab_setup_table_t setup;
mfn_t frames[NR_GRANT_FRAMES];

setup.dom = DOMID_SELF;
setup.nr_frames = NR_GRANT_FRAMES;
// get frame list from Xen
// (defined in xen/include/public/arch-x86/xen.h)
set_xen_guest_handle(setup.frame_list, frames);

HYPERVISOR_grant_table_op(GNTTABOP_setup_table, &setup, 1);
// map into page table
gnttab_table = map_frames(frames, NR_GRANT_FRAMES);
printk("gnttab_table mapped at %p.\n", gnttab_table);
```


Granting a shared paged

```
// after grant table made visible to Xen,
// grant pages can be shared
grant_ref_t
gnttab_grant_access(domid_t domid, ulong frame, int readonly)
{
  grant_ref_t ref = get_free_entry();
  gnttab_table[ref].frame = frame;
  gnttab_table[ref].domid = domid;
  wmb();
  readonly *= GTF_readonly;
  gnttab_table[ref].flags = GTF_permit_access | readonly;

  return ref;
}
```


Part XIV

XenStore/XenBus

XenStore

Systems need a way to “discover” new things

- In user space, this is accomplished by file system or by network services
- For example, some information can be put in a certain directory by the “advertiser”
- And the directory can be periodically read by a process interested in such services
- In Xen, the equivalent is the XenStore
- XenStore is a hierarchically structured file system, it stores name,value pairs
- XenBus is the means of communicating with the XenStore

XenStore semantics

- Read a Key
- Write a Key
- Notify when a Key changes
- Iterator through a directory
- It supports transactions (and thus atomic operations)
- The XenStore is also accesible from user space, so that Dom0 tools can be used to manipulate them

XenStore layout

- Domain specific information is stored by UUID, a "universally unique ID"
- UUID are essentially long random numbers, and hence the chance of a conflict (two domains using the same UUID) is essentially nil
- `/vm/`*uuid* stores configuration information about the domain with universal *uuid*
- `/local/domain/`*uuid*
- see `http://wiki.xensource.com/xenwiki/XenStoreReference`

XenStore commands

```
XS_READ,          // read a key
XS_WATCH,         // watch for changes to key-value
XS_UNWATCH,       // unwatch a previously watched key
                 // multi-message transaction
XS_TRANSACTION_START, // start the transaction
XS_TRANSACTION_END,   // end the transaction
XS_WRITE,          // write a key
XS_WATCH_EVENT,// response to a watch
XS_ERROR,          // an error
```


XenStore requests/responses

```
struct xsd_sockmsg {
uint32_t type;   // XS_???
uint32_t req_id;// Request ID,
                 // echoed in daemon's response.
uint32_t tx_id;  // Transaction id
                 // (0 if not in a transaction).
uint32_t len;    // Length of data following this.
};
```

- The xsd_sockmsg is followed by zero or more null terminated strings
- The xsd_sockmsg plus following strings must fit within the 1024 byte buffer
- Errors are returned as strings, using type XS_ERROR

XenStore buffers

```
struct xenstore_domain_interface {
 char req[XENSTORE_RING_SIZE]; // XenStore requests
 char rsp[XENSTORE_RING_SIZE]; // XenStore replies
 XENSTORE_RING_IDX req_cons, req_prod;
 XENSTORE_RING_IDX rsp_cons, rsp_prod;
};
```

- Two 1K buffers
- One each for requests, responses to the XenStore
- The normal producer/consumer indexes

Initializing the XenStore

```
int xenstore_init(start_info_t * start)
{
  ulong pfn = machine_to_phys_mapping[
                          start->store_mfn];
  xenstore = (struct xenstore_domain_interface*)
     ((pfn << 12) + ((ulong)&_text));
  xenstore_evt = start->store_evtchn;
  // TODO: Set up the event channel

  return 0;
}
```


XenStore write

```
static uint req_id = 0; // incremented for each req

int xenstore_write(char * key, char * value)
{
  int key_length = strlen(key) + 1;
  int value_length = strlen(value) + 1;
  struct xsd_sockmsg msg = {
    .type = XS_WRITE,
    .req_id = req_id,
    .tx_id = 0,
    .len = key_length + value_length};
  // Write the message
  xenstore_write_request((char*)&msg, sizeof(msg));
  xenstore_write_request(key, key_length);
  xenstore_write_request(value, value_length);
```


XenStore write

```
  // Notify the back end
  NOTIFY();

  // really should do more error processing
  xenstore_read_response((char*)&msg, sizeof(msg));
  IGNORE(msg.len);
  if (msg.req_id != req_id++)
  {
    return -1;
  }
  return 0;
}
```

Write the request

```
int xenstore_write_request(char *message, int lengt
{
  // Check that the message will fit
  if (length > XENSTORE_RING_SIZE) { return -1; }

  XENSTORE_RING_IDX i; // Fixed bug in original code

  // write bytes to ring buffer (see next slide)

  // Data is written to the ring, make it visible
  wmb();
  xenstore->req_prod = i; // now tell Dom0 about it
  return 0;
}
```


Write request inner loop

```
for (i=xenstore->req_prod; length > 0; i++,length--
 {
   // Wait for the back end to clear enough space i
   XENSTORE_RING_IDX data;
   do
   {
     data = i - xenstore->req_cons;
     mb();
   } while (data >= sizeof(xenstore->req));
   // Copy the byte
   int ring_index = MASK_XENSTORE_IDX(i);
   xenstore->req[ring_index] = *message;
   message++;
 }
```


Some macros

```
#define NOTIFY() \
       do\
       {\
         struct evtchn_send event;\
         event.port = xenstore_evt;\
         HYPERVISOR_event_channel_op(\
                EVTCHNOP_send, &event);\
       } while(0)

#define IGNORE(n) \
        do\
        {\
          char buffer[XENSTORE_RING_SIZE];\
          xenstore_read_response(buffer, n);\
        } while(0)
```


Part XV

Device drivers

Split drivers

- Xen doesn't contain device drivers
- Instead it relies on those in Dom0
- Hence Dom0 has privileges to access device drivers (DomU's don't)
- In fact, most code in Dom0 is device driver code
- Xen exports a number of pseudo-devices, including a console, disk, network.
- Xen's device interfaces are not like other VMs device drivers
- Which typically export qemu simulated devices
- In either event, the exported devices are independent of the underlying physical devices

Front end/back end

- Xen's devices are exported as ring buffers implemented in shared memory
- they are paired with events for signaling data availability
- A DomU implements the front-end device which
  - Puts the data in the shared memory buffer
  - And then uses a Xen event to signal that there is data available
- A Dom0 implements the back-end device which on output
  - Has a handler associated with the front-end device
  - When it receives an event, it remove the data from the ring buffer
  - And schedules it to be written out
  - Using Dom0 device drivers for that device
- On input the direction is from Dom0 to DomU

Part XVI

Concurrency

Concurrency issues

- Concurrency issues are minimized in nanoOS because only one processor is executing in the OS at a time.
- But concurrency is inherent in OSs
- The primary issues we need to deal with are
  - Interrupts/Events can occur when other operations are occurring.
  - Memory semantics, in which ordering of memory operations is not consistent across cores.
  - Waits on shared structures for external events.

Concurrency issues

Interrupts Minimize upper half processing. Disable interrupts when doing the lower half processing.

Memory use barrier to ensure that memory operations are ordered relative to each other when using shared memory.

Waits latch object and queue if object is already busy. Must ensure deadlock is avoided.

Atomic compare and exchange.

Part XVII

Disk

Split driver model

- Xen uses a split driver model in which I/O is performed in two steps
  - DomU requests I/O from Dom0
  - Dom0 performs the physical I/O using Linux's device drivers
- The DomU code is called the front end driver
- The Dom0 code is called the back end driver
  - communicates with the back end
  - contains the device drivers for the physical devices
- This leverages the enormous codebase of device drivers in Linux
- And simplifies DomU device handling and Xen code

Rings

- Front end communicates with the back end using rings
- It does this via ring indices which are uints
- each index is initialized to zero and incremented only by one side
- Requests are generated by one side and responses by the other
- The disk drive is implemented with one ring, since all disk operations are in response to DomU requests.

Device driver rings

Req_0^u				DomU puts on request 0

Device driver rings

Req_0^u				DomU puts on request 0
Req_0^u	Req_1^u			DomU puts on request 1

Device driver rings

Req_0^u				DomU puts on request 0
Req_0^u	Req_1^u			DomU puts on request 1
Res_0^0	Req_1^u			Dom0 responds to request 0

Device driver rings

Req_0^u				DomU puts on request 0
Req_0^u	Req_1^u			DomU puts on request 1
Res_0^0	Req_1^u			Dom0 responds to request 0
	Req_1^u			DomU removes response 0

Device driver rings

Req_0^u				DomU puts on request 0
Req_0^u	Req_1^u			DomU puts on request 1
Res_0^0	Req_1^u			Dom0 responds to request 0
	Req_1^u			DomU removes response 0
	Res_1^0			Dom0 responds to request 1

Device driver rings

Req_0^u				DomU puts on request 0
Req_0^u	Req_1^u			DomU puts on request 1
Res_0^0	Req_1^u			Dom0 responds to request 0
	Req_1^u			DomU removes response 0
	Res_1^0			Dom0 responds to request 1
				DomU removes response 1

There are three indices in this scheme, all initially 0

- **start** incremented by DomU when response is removed.
- **tail** points one after the last request enqueued. Incremented by DomU.
- **response** is between start and tail and is the last response added. Incremented by Dom0.

A Disk driver

- The ring buffer is just for disk command
- The data is transfered in separate grant pages
- Data is transferred in 512 byte sectors,
- The device might require 4K blocks
- A disk op can ask for multiple segments to be read or written
- Disk ops can be reordered by backend, and hence there is a disk barrier

Disk driver initialization

```
blkif_string_t *ring_shared;
blkif_front_ring_t ring_private;

int init_disk(void)
{
  shared = new_page(); // allocate a page
  SHARED_RING_INIT(shared_page);
  FRONT_RING_INIT(&ring_private,
                  ring_shared, PAGE_SIZE);
  // get available grant ref
  grant_entry_t *ref = get_grant_ref();
  ref->frame = virt_to_mfn(shared);
  ref->domid = backend_domain;
  wmb();
  ref->flags = GTF_permit_access; // Book bug
}
```

Block driver request

```
#define BLKIF_OP_READ              0
#define BLKIF_OP_WRITE             1
#define BLKIF_OP_WRITE_BARRIER     2
#define BLKIF_OP_FLUSH_DISKCACHE   3

// sectors are 512 bytes.
// normally, first_sect =0 and last_sect=7
struct blkif_request_segment {
  grant_ref_t gref; // reference to I/O buffer fram
  // first_sect: first sector in frame to transfer
  // last_sect: last sector in frame to transfer
  uint8_t     first_sect, last_sect;
};
```


Block driver request

```
struct blkif_request {
  uint8_t        operation;    // BLKIF_OP_???
  uint8_t        nr_segments;  // number of segment
  uint64_t       id;           // private guest val
  blkif_vdev_t   handle;       // only for read/wri
  blkif_sector_t sector_number;// start sector idx
  struct blkif_request_segment
               seg[BLKIF_MAX_SEGMENTS_PER_REQUEST]
};
```


Reading a block

```
  char *page = new_page(); // Book Bug
  grant_table_entry_t *ref  = get_grant_ref();
  ref->frame = virt_to_mfn(page);
  ref->domid = backend_domain;
  wmb();
  ref->flags = GTF_permit_access;

int readBlock(grant_table_entry_t *ref,
 blkif_sector_t  sector,   // disk sector address
 uint64          requestId, // echoed back on respons
 uint        shouldNotify); // send event to do the
```


readBlock

```
blkif_request_t * request =
  RING_GET_REQUEST(private, private->req_prod++);
request->operation  = BLKIF_OP_READ;
request->handle      = block_vdev;
request->sector_number = sector; // on disk
request->id = requestId;
request->nr_segments = 1;
request->seg[0].gref = ref - GRANT_TABLE;
request->seg[0].first_sect = 0;
request->seg[0].last_sect  = 7;
RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(
  private, shouldNotify);
if (shouldNotify) {
  struct evtchn_send event;
  event.port = block_port;
  HYPERVISOR_event_channel_op(EVTCHNOP_send, &event
```

Response from read

```
struct blkif_response {
  uint64_t        id;        // copied from request
  uint8_t         operation; // copied from request
  int16_t         status;    // BLKIF_RSP_???
};

// i is the index into the ring buffer
blkif_response_t *response =
    RING_GET_RESPONSE(private_ring, i);
if (response->id == requestId)
{
}
```


Completing back end initialization

- The initialization given so far is for the front end
- But still need to configure the back end
- To do that, the back end needs
  - The machine address of the ring page
  - The event channel for the device
- The front end needs to negotiate with the backend
- And perhaps get some information from the backend
- Such as device characteristics
- All these are done using the XenBus

XenBus calls

- xenbus_transaction_start (&xbt) starts a transaction, reports back with transaction ID
- xenbus_printf (xbt, dir, key, fmt, v) writes to the XenStore in dir the key is given the value which is the result of sprintf (str, fmt, v)
- xenbus_switch_state (xbt, path, XenbusStateConnected), then the device is changed to the connected state. (It does this by doing a XenStore write of the new state for the path)
- xenbus_transaction_end (xbt, 0, &retry) ends the transaction

XenBus setup

```
err = xenbus_transaction_start(&xbt);

err = xenbus_printf(xbt, nodename, "ring-ref","%u",
                    dev->ring_ref);

err = xenbus_printf(xbt, nodename,
              "event-channel", "%u", dev->evtchn);

err = xenbus_printf(xbt, nodename, "protocol",
              "%s", XEN_IO_PROTO_ABI_NATIVE);

snprintf(path, sizeof(path), "%s/state", nodename);
err = xenbus_switch_state(xbt, path,
                 XenbusStateConnected);

err = xenbus_transaction_end(xbt, 0, &retry);
```

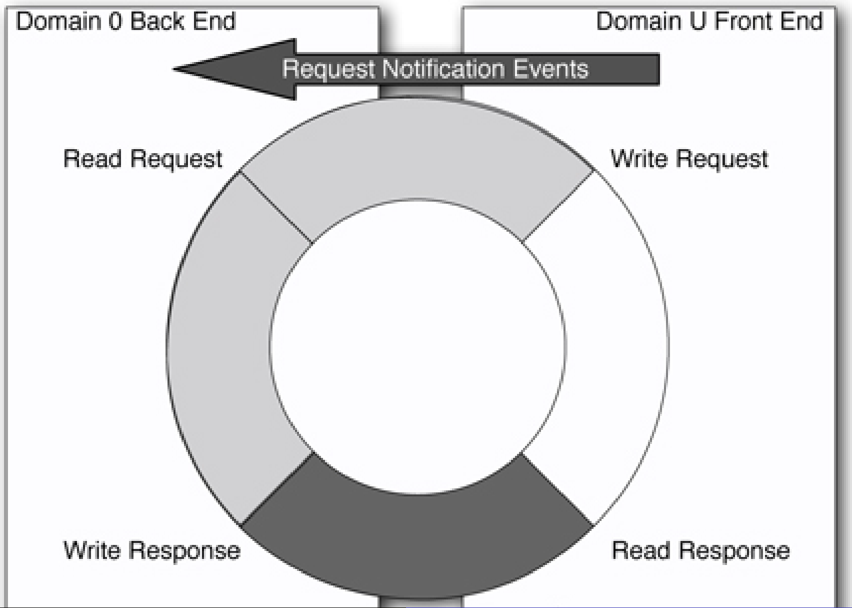
Part XVIII

Networking

Networking

- A network interface is implemented with two rings, one for arriving packets and one for outgoing packets
- This is because packet do not always arrive in response to a request
- And even if they did, the latency is too large to hold up other packets
- The disk driver's data was disk blocks
- The network driver's data is ethernet packets
- And thus ethernet, IP headers, TCP or UDP headers need to be added

Networking



Part XIX

Scheduling

Scheduling

- In a bare metal OS, the OS can
  - Halt (stop running)
  - Can go into low power state
  - Can block waiting for an interrupt
- On a multi-core processor, different processors can be executing simultaneously in Dom0, DomU, and Xen.
- On a uniprocessor, DomU should help by giving up the processor if it needs to wait for Dom0 to do something

```
HYPERVISOR_sched_op(SCHED_OP_yield, NULL);
```


Halting and Blocking

- to shutdown

```
sched_shutdown_op_t op;
op.reason = SHUTDOWN_power_off;
HYPERVISOR_sched_op(SCHED_OP_shutdown, &op);
```

- give up processor until next event delivered

```
HYPERVISOR_sched_op(SCHED_OP_block, &op);
```


Part XX

Summary

Summary

- A VM abstracts the hardware
- All VMs abstract I/O devices on PCs because the variety of I/O devices is so large that they are too large to implement
- Xen's paravirtualization abstraction also abstracts the architecture
  - by starting up in real mode
  - and by requiring hypercalls instead of privileged instructions
- Xen's paravirtualization was designed to be very close to the actual hardware
- To minimize changes when porting to Xen
- Since the time Xen was introduced, Intel and AMD have provided virtualization extensions for their architecture
- Enabling unmodified kernels to be run under Xen

Summary

- In addition, interrupts are supported for entering into the kernel
- Events are the abstraction of interrupts
- They can be used for physical IRQs in privileged domains, virtual IRQs, and interdomain communication.
- These events can be associated with handlers which are asynchronously called
- But this requires synchronization to prevent race conditions
- And it is best to minimize the amount of work done in the upper handler

Bringing the OS up

- Xen boots with paging turned on
- It initially communicates with the OS through start_info and shared_info
  - Which provides handles necessary to set up XenBus and console
  - And information about the virtual CPUs allocated to the OS
  - And event delivery from Xen to the OS
  - And time mechanisms for wallclock and scheduling
- With this information the console and XenStore can be brought up

Bringing the OS up (cont'd)

- The Grant Table needs to be established, providing pages which can be shared between domains.
- A time of day clock is created which get timer interrupts
  - Timers ensure that a process does not hog the CPU
  - (The kernel is coded in such a way that this is not a danger)
- The XenStore is then brought up, allowing DomU to request new services from Dom0

Disk and Network Devices

- Devices depend on front-end drivers in DomU communicating with back-end Dom0 drivers
- The front end
  - Creates a ring buffer using a GrantTable Page
  - An interdomain event port is created to signal when data is put on or taken off the ring buffer.
  - This information is communicated with the backend via the XenBus
- The back end then configures the device
- GrantTable pages are then used for sending data to/from the back end