Lguest
the little hypervisor

Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

based on lguest documentation
written by Rusty Russell

note: this is a draft and incomplete version

lguest

- introduction
- launcher
- guest kernel
- host kernel
- switcher

don't have time for:

- virtio
- virtual devices
- patching technique
- async hypercalls
- rewriting technique
- guest virtual interrupt controller
- guest virtual clock
- hw assisted vs paravirtualization
- hybrid virtualization
- benchmarks

introduction

- A hypervisor allows multiple Operating Systems to run on a single machine.
- lguest
  - paravirtualization within linux kernel
  - proof of concept (paravirt_ops, virtio...)
  - teaching tool

guests and hosts

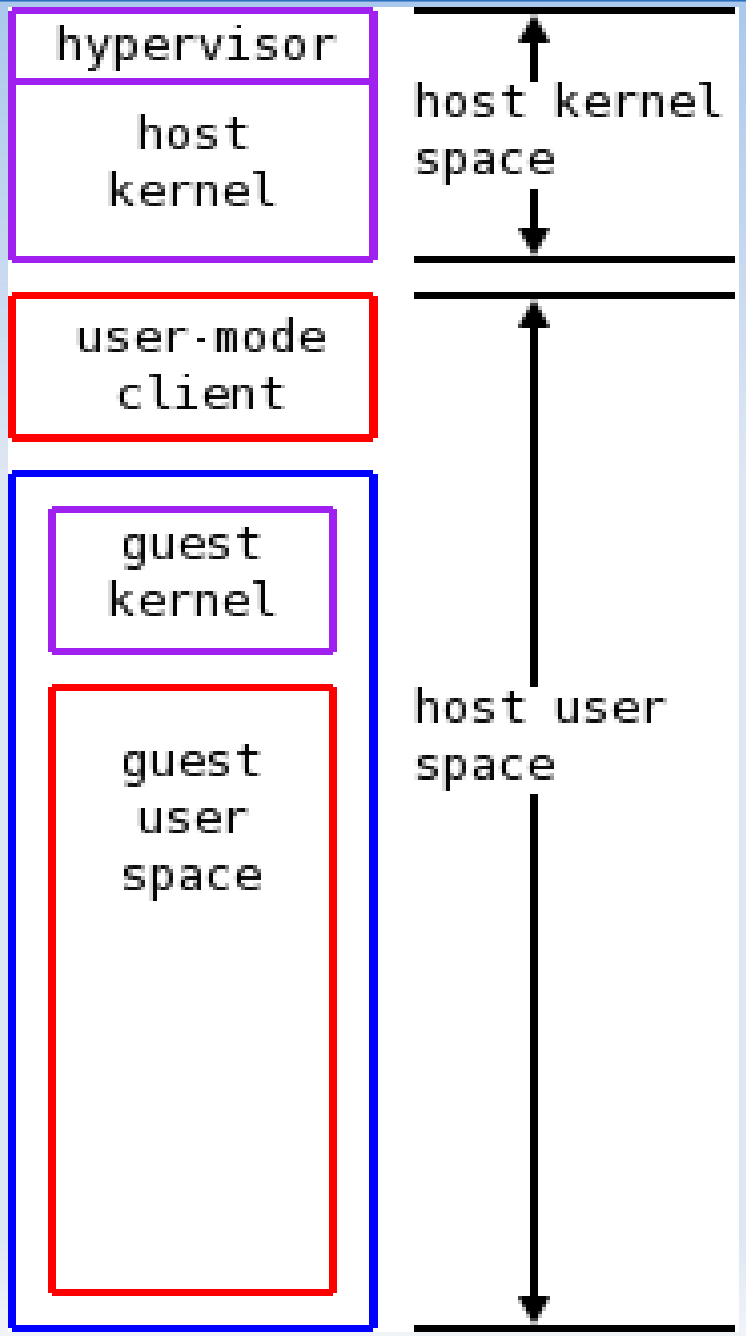
- A **module** (lg.ko) allows us to **run other Linux kernels** the same way we'd run processes.

- We only run specially **modified Guests.** Setting CONFIG_LGUEST_GUEST, compiles [arch/x86/lguest/boot.c] into the kernel so **it knows how to be a Guest at boot time**. This means that you can use the same kernel you boot normally (ie. as a Host) as a Guest.

- These Guests know that they **cannot do privileged operations**, such as disable interrupts, and that they have to ask the Host to do such things explicitly.

- Some of the replacements for such low-level native hardware operations call the Host through a "**hypercall**". [drivers/lguest/hypercalls.c]

high level overview



- launcher
- /dev/lguest
- guest kernel
- host kernel
- switcher

Launcher

- main()
- some implementations
- /dev/lguest
- write → initialize
- read → run guest

Launcher main [Documentation/lguest/lguest.c]

The Launcher is the **Host userspace program which sets up, runs and services the Guest**. Just to confuse you: **to the Host kernel, the Launcher *is* the Guest** and we shall see more of that later.

Guest physical == Launcher virtual with an offset.

```
main {
        mem = atoi(argv[i]) * 1024 * 1024;

        /* We start by mapping anonymous pages over all of guest-physical memory
        * range.  This fills it with 0, and ensures that the Guest won't be killed when
        *  it tries to access it. */
        guest_base = map_zeroed_pages(mem / getpagesize() + DEVICE_PAGES);
        guest_limit = mem;
        guest_max = mem + DEVICE_PAGES*getpagesize();

static void *map_zeroed_pages(unsigned int num)
{
......
        addr = mmap(NULL, getpagesize() * num,
                PROT_READ|PROT_WRITE|PROT_EXEC, MAP_PRIVATE, fd, 0);
......
```

Launcher main

```
/* Now we load the kernel */
start = load_kernel(open_or_die(argv[optind+1], O_RDONLY));

/* Boot information is stashed at physical address 0 */
boot = from_guest_phys(0);

/* Map the initrd image if requested (at top of physical memory) */
if (initrd_name) {
	initrd_size = load_initrd(initrd_name, mem);

	/* These are the location in the Linux boot header where the
	 * start and size of the initrd are expected to be found. */
	boot->hdr.ramdisk_image = mem - initrd_size;
	boot->hdr.ramdisk_size = initrd_size;

	/* The bootloader type 0xFF means "unknown"; that's OK. */
	boot->hdr.type_of_loader = 0xFF;
}
```

Launcher main

```
/* Set up the initial linear pagetables, starting below the initrd. */
pgdir = setup_pagetables(mem, initrd_size);

/* The Linux boot header contains an "E820" memory map: ours is a
 * simple, single region. */
boot->e820_entries = 1;
boot->e820_map[0] = ((struct e820entry) { 0, mem, E820_RAM });

/* The boot header contains a command line pointer: we put the command
 * line after the boot header. */
boot->hdr.cmd_line_ptr = to_guest_phys(boot + 1);

/* We use a simple helper to copy the arguments separated by spaces. */
concat((char *)(boot + 1), argv+optind+2);

/* Boot protocol version: 2.07 supports the fields for lguest. */
boot->hdr.version = 0x207;
```

Launcher main

```
/* The hardware_subarch value of "1" tells the Guest it's an lguest. */
boot->hdr.hardware_subarch = 1;

.......

/* We tell the kernel to initialize the Guest: this returns the open
 * /dev/lguest file descriptor. */
lguest_fd = tell_kernel(pgdir, start);


/* We clone off a thread, which wakes the Launcher whenever one of the
 * input file descriptors needs attention.  We call this the Waker */
setup_waker(lguest_fd);

/* Finally, run the Guest.  This doesn't return. */
run_guest(lguest_fd);
```

Launcher & /dev/lguest

We begin our understanding with the **Host kernel interface which the Launcher uses**: reading and writing a character device called **/dev/lguest**.

All the work happens in the read(), write() and close() routines:

```
static struct file_operations lguest_fops = {
        .owner   = THIS_MODULE,
        .release = close,
        .write   = write,
        .read    = read,
};
```

the waker

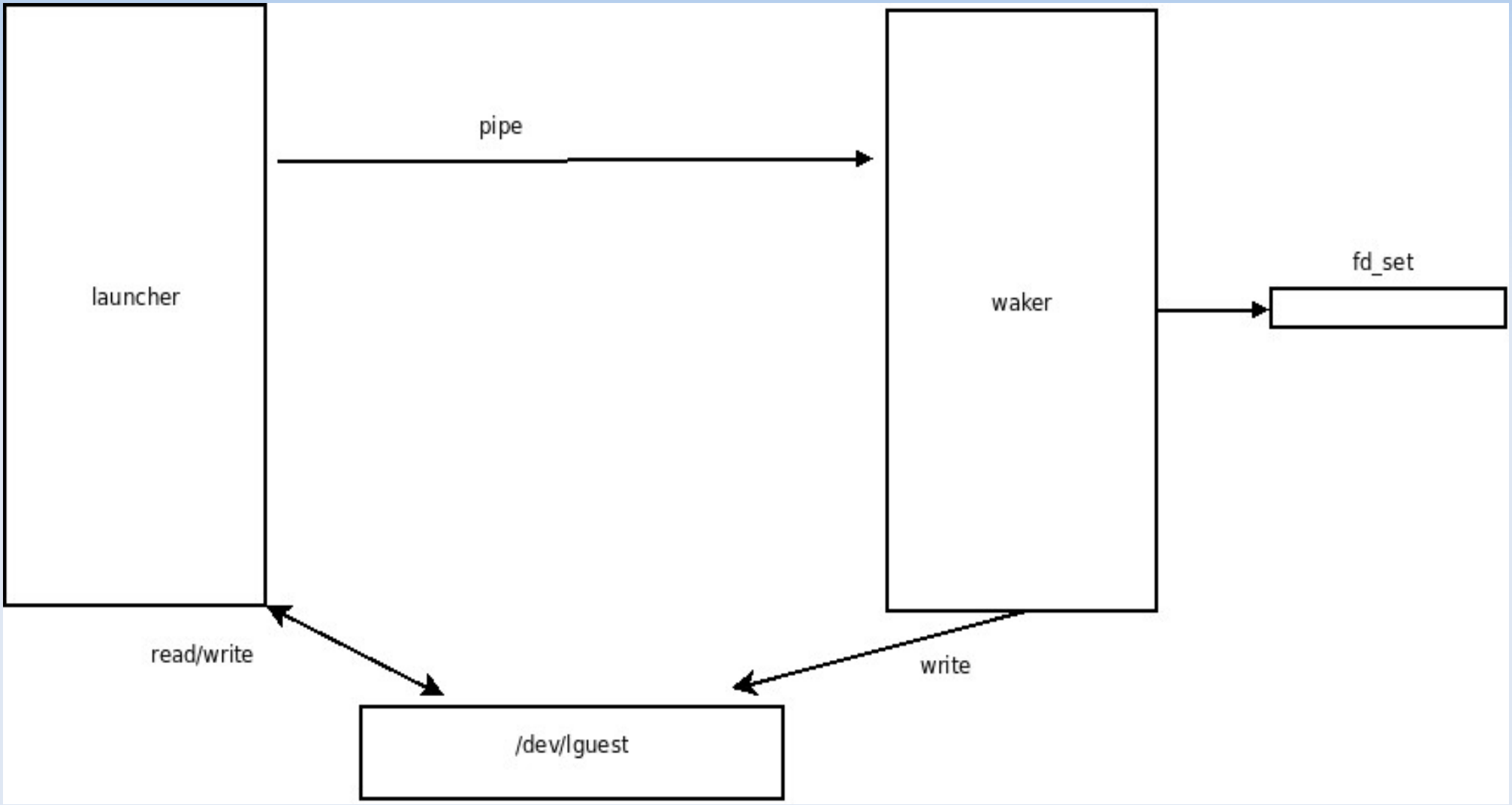
With console, block and network devices, we can have lots of input which we need to process. We could try to tell the kernel what file descriptors to watch, but **handing a file descriptor mask through to the kernel is fairly icky.**

Instead, we clone off **a thread which watches the file descriptors and writes the LHREQ_BREAK command to the /dev/lguest file descriptor to tell the Host stop running the Guest**.

This causes the Launcher to **return from the /dev/lguest read with -EAGAIN**, where it will write to /dev/lguest to reset the LHREQ_BREAK and wake us up again.

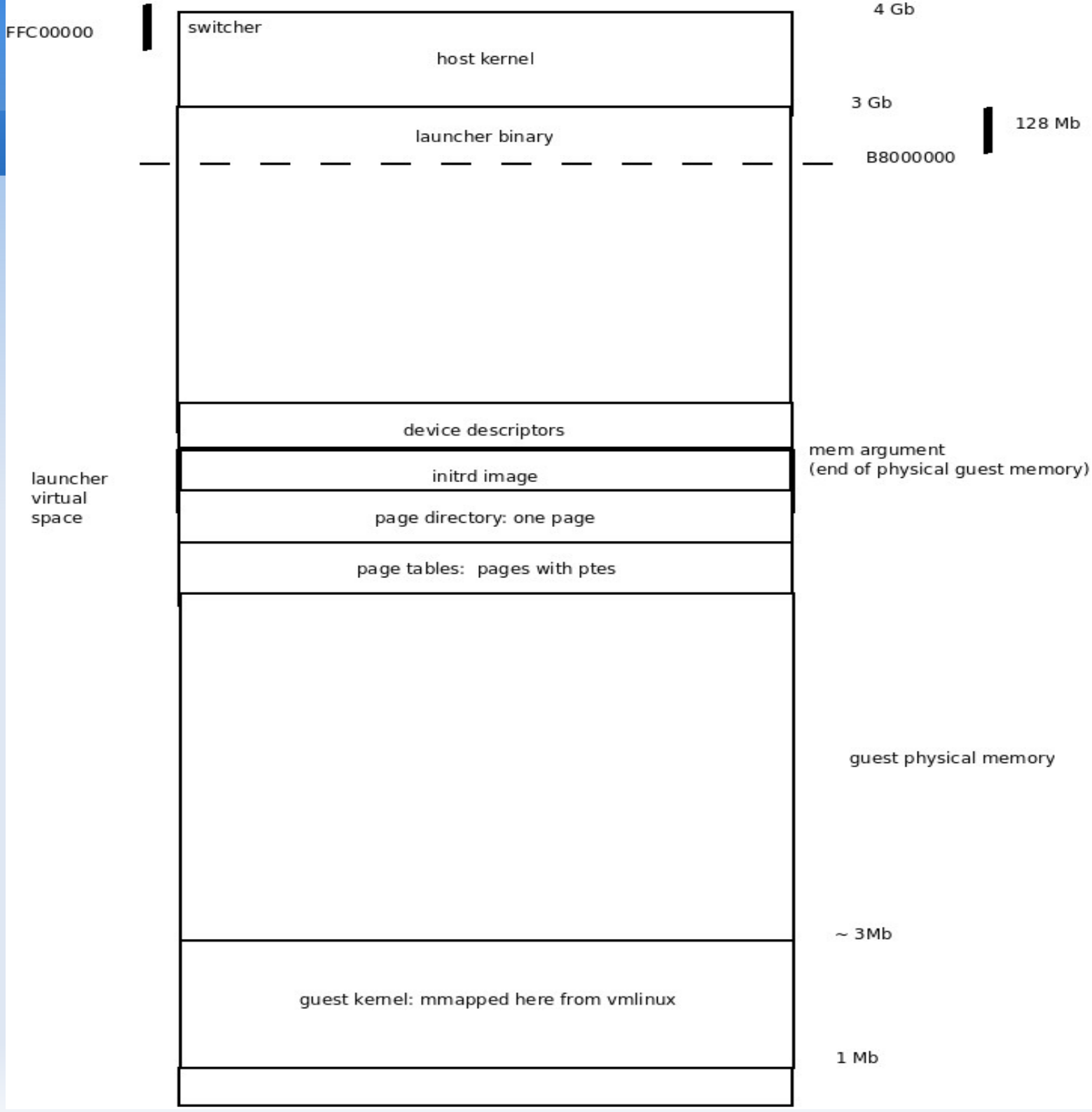
This, of course, is merely a different *kind* of icky.

launcher, waker and lg.ko



some implementations: setup_pagetables()

```
static unsigned long setup_pagetables(unsigned long mem, unsigned long initrd_size)
{
        unsigned long *pgdir, *linear;          mapped_pages = mem/getpagesize();

        /* Each PTE page can map ptes_per_page pages: how many do we need? */
        linear_pages = (mapped_pages + ptes_per_page-1)/ptes_per_page;

        /* We put the toplevel page directory page at the top of memory. */
        pgdir = from_guest_phys(mem) - initrd_size - getpagesize();

        /* Now we use the next linear_pages pages as pte pages */
        linear = (void *)pgdir - linear_pages*getpagesize();

        /* Linear mapping is easy: put every page's address into the mapping in
         * order.  PAGE_PRESENT contains the flags Present, Writable and
         * Executable. */
        for (i = 0; i < mapped_pages; i++)
                linear[i] = ((i * getpagesize()) | PAGE_PRESENT);

        /* The top level points to the linear page table pages above. */
        for (i = 0; i < mapped_pages; i += ptes_per_page) {
                pgdir[i/ptes_per_page] = ((to_guest_phys(linear)
                                                + i*sizeof(void *)) | PAGE_PRESENT);
        }
```


4 Gb
FFC00000
switcher
host kernel
3 Gb
128 Mb
launcher binary
B8000000
device descriptors
mem argument
(end of physical guest memory)
launcher
virtual
space
initrd image
page directory: one page
page tables: pages with ptes
guest physical memory
~ 3Mb
guest kernel: mmapped here from vmlinux
1 Mb

some implementations: tell_kernel()

```
/* This is where we actually tell the kernel to initialize the Guest.  We
 * saw the arguments it expects when we looked at initialize() in lguest_user.c:
 * the base of Guest "physical" memory, the top physical page to allow, the
 * top level pagetable and the entry point for the Guest. */

static int tell_kernel(unsigned long pgdir, unsigned long start)
{
        unsigned long args[] = { LHREQ_INITIALIZE,
                        (unsigned long)guest_base,
                        guest_limit / getpagesize(), pgdir, start };
        int fd;

        fd = open_or_die("/dev/lguest", O_RDWR);
        if (write(fd, args, sizeof(args)) < 0)
              err(1, "Writing to /dev/lguest");

        /* We return the /dev/lguest file descriptor to control this Guest */
        return fd;
}
```

write()

The first operation the Launcher does must be a write. All writes start with an unsigned long number: **for the first write this must be LHREQ_INITIALIZE to set up the Guest**.

After that the Launcher can use writes of other values to **send interrupts**.

```
static ssize_t write(struct file *file, const char __user *in,
                     size_t size, loff_t *off)
{
....
        switch (req) {
        case LHREQ_INITIALIZE:
                return initialize(file, input);
        case LHREQ_IRQ:
                return user_send_irq(cpu, input);
        case LHREQ_BREAK:
                return break_guest_out(cpu, input);
```

initialize()

The initialization write supplies 4 pointer sized (32 or 64 bit) values. These are:

base: The start of the Guest-physical memory inside the Launcher memory.
pfnlimit: The highest (Guest-physical) page number the Guest should be allowed to access.
pgdir: The (Guest-physical) address of the top of the initial Guest pagetables (which are set up by the Launcher).
start: The first instruction to execute ("eip" in x86-speak).

```
static int initialize(struct file *file, const unsigned long __user *input)
{
...
        /* Populate the easy fields of our "struct lguest" */
        lg->mem_base = (void __user *)args[0];
        lg->pfn_limit = args[1];

        /* This is the first cpu (cpu 0) and it will start booting at args[3] */
        err = lg_cpu_start(&lg->cpus[0], 0, args[3]);

        /* Initialize the Guest's shadow page tables, using the toplevel
         * address the Launcher gave us.  This allocates memory, so can fail. */
        err = init_guest_pagetable(lg, args[2]);
```

lg_cpu_start

```
static int lg_cpu_start(struct lg_cpu *cpu, unsigned id, unsigned long start_ip)
{
        ....
        /* We need a complete page for the Guest registers: they are accessible
         * to the Guest and we can only grant it access to whole pages. */
        cpu->regs_page = get_zeroed_page(GFP_KERNEL);

        /* We actually put the registers at the bottom of the page. */
        cpu->regs = (void *)cpu->regs_page + PAGE_SIZE - sizeof(*cpu->regs);

        /* Now we initialize the Guest's registers, handing it the start
         * address. */
        lguest_arch_setup_regs(cpu, start_ip);
```

lguest_arch_setup_regs()

```
void lguest_arch_setup_regs(struct lg_cpu *cpu, unsigned long start)
{
        struct lguest_regs *regs = cpu->regs;

        regs->ds = regs->es = regs->ss = __KERNEL_DS|GUEST_PL;
        regs->cs = __KERNEL_CS|GUEST_PL;

      /*  We always leave interrupts enabled while running the Guest. */
        regs->eflags = X86_EFLAGS_IF | 0x2;

        regs->eip = start;

        /* %esi points to our boot information, at physical address 0, so don't
         * touch it. */

        /* There are a couple of GDT entries the Guest expects when first
         * booting. */
        setup_guest_gdt(cpu);
}
```

setup_guest_gdt()

```
/* This routine sets up the initial Guest GDT for booting.  All entries start
 * as 0 (unusable). */
void setup_guest_gdt(struct lg_cpu *cpu)
{
	/* Start with full 0-4G segments... */
	cpu->arch.gdt[GDT_ENTRY_KERNEL_CS] = FULL_EXEC_SEGMENT;
	cpu->arch.gdt[GDT_ENTRY_KERNEL_DS] = FULL_SEGMENT;

	/* ...except the Guest is allowed to use them, so set the privilege
	 * level appropriately in the flags. */
	cpu->arch.gdt[GDT_ENTRY_KERNEL_CS].b |= (GUEST_PL << 13);
	cpu->arch.gdt[GDT_ENTRY_KERNEL_DS].b |= (GUEST_PL << 13);
}
```

run_guest (launcher)

```
/* Finally we reach the core of the Launcher which runs the Guest, serves
 * its input and output, and finally, lays it to rest. */

static void __attribute__((noreturn)) run_guest(int lguest_fd)
{
        for (;;) {
                unsigned long args[] = { LHREQ_BREAK, 0 };
                unsigned long notify_addr;
                int readval;

                /* We read from the /dev/lguest device to run the Guest. */
                readval = pread(lguest_fd, &notify_addr,
                                sizeof(notify_addr), cpu_id);

                /* One unsigned long means the Guest did HCALL_NOTIFY */
                if (readval == sizeof(notify_addr)) {
                        verbose("Notify on address %#lx\n", notify_addr);
                        handle_output(lguest_fd, notify_addr);
                        continue;
```

cont: run_guest (launcher)

```
        /* ENOENT means the Guest died.  Reading tells us why. */
        } else if (errno == ENOENT) {
                char reason[1024] = { 0 };
                pread(lguest_fd, reason, sizeof(reason)-1, cpu_id);
                errx(1, "%s", reason);
        /* ERESTART means that we need to reboot the guest */
        } else if (errno == ERESTART) {
                restart_guest();
        /* EAGAIN means a signal (timeout).
         * Anything else means a bug or incompatible change. */
        } else if (errno != EAGAIN)
                err(1, "Running guest failed");

        /* Only service input on thread for CPU 0. */
        if (cpu_id != 0)
                continue;

        /* Service input, then unset the BREAK to release the Waker. */
        handle_input(lguest_fd);
        if (pwrite(lguest_fd, args, sizeof(args), cpu_id) < 0)
                err(1, "Resetting break");
}
```

read() [drivers/lguest/lguest_user.c]

Once our Guest is initialized, the Launcher makes it run by reading from /dev/lguest.

```
static ssize_t read(struct file *file, char __user *user, size_t size,loff_t*o)
{
        .....

        /* If we returned from read() last time because the Guest sent I/O,
         * clear the flag. */
        if (cpu->pending_notify)
              cpu->pending_notify = 0;

        /* Run the Guest until something interesting happens. */
        return run_guest(cpu, (unsigned long __user *)user);
}
```

guest kernel

- initialization
- paravirt ops
- hypercalls
- some implementations of paravirt ops

Initialization

- Our story starts with the kernel booting into startup_32 in arch/x86/kernel/head_32.S.
- It expects a boot header, which is created by the bootloader (the Launcher in our case).
- The startup_32 function does very little: it clears the uninitialized global C variables which we expect to be zero (ie. BSS) and then copies the boot header and kernel command line somewhere safe.
- Finally it checks the 'hardware_subarch' field. This was introduced in 2.6.24 for lguest and Xen: if it's set to '1' (lguest's assigned number), then it calls us (lguest_entry).

ENTRY(lguest_entry)

[arch/x86/lguest/i386_head.S]

```
/* We make the "initialization" hypercall now to tell the Host about us, and also find out where
it put our page tables. */
movl $LHCALL_LGUEST_INIT, %eax
movl $lguest_data - __PAGE_OFFSET, %edx
int $LGUEST_TRAP_ENTRY

/* The Host put the toplevel pagetable in lguest_data.pgdir.  The movsl instruction uses %esi
implicitly as the source for the copy we're about to do. */
movl lguest_data - __PAGE_OFFSET + LGUEST_DATA_pgdir, %esi

/* Copy first 32 entries of page directory to __PAGE_OFFSET entries. This means the first
 128M of kernel memory will be mapped at PAGE_OFFSET where the kernel expects to run.
 This will get it far enough through boot to switch to its own pagetables. */
movl $32, %ecx
movl %esi, %edi
addl $((__PAGE_OFFSET >> 22) * 4), %edi
rep
movsl

/* Set up the initial stack so we can run C code. */
movl $(init_thread_union+THREAD_SIZE),%esp

/* Jumps are relative, and we're running __PAGE_OFFSET too low at the moment. */
jmp lguest_init+__PAGE_OFFSET
```

[arch/x86/lguest/boot.c]

- Once we get to **lguest_init()**, we know we're a Guest. The various pv_ops structures in the kernel provide points for (almost) every routine we have to override to avoid privileged instructions.

- We replace the native functions with our Guest versions, then boot like normal.

lguest_init(void)
[arch/x86/lguest/boot.c]

```
/* interrupt-related operations */
pv_irq_ops.init_IRQ = lguest_init_IRQ;
pv_irq_ops.save_fl = save_fl;
pv_irq_ops.restore_fl = restore_fl;
pv_irq_ops.irq_disable = irq_disable;
pv_irq_ops.irq_enable = irq_enable;
pv_irq_ops.safe_halt = lguest_safe_halt;

/* init-time operations */
pv_init_ops.memory_setup = lguest_memory_setup;
pv_init_ops.patch = lguest_patch;

/* Intercepts of various cpu instructions */
pv_cpu_ops.load_gdt = lguest_load_gdt;
pv_cpu_ops.cpuid = lguest_cpuid;
pv_cpu_ops.load_idt = lguest_load_idt;
pv_cpu_ops.iret = lguest_iret;
pv_cpu_ops.load_sp0 = lguest_load_sp0;
pv_cpu_ops.load_tr_desc = lguest_load_tr_desc;
pv_cpu_ops.set_ldt = lguest_set_ldt;

/* pagetable management */
pv_mmu_ops.write_cr3 = lguest_write_cr3;
pv_mmu_ops.flush_tlb_user = lguest_flush_tlb_user;
pv_mmu_ops.flush_tlb_single = lguest_flush_tlb_single;
pv_mmu_ops.flush_tlb_kernel = lguest_flush_tlb_kernel;
pv_mmu_ops.set_pte = lguest_set_pte;
```

hypercalls

- But first, how does our Guest contact the Host to ask for privileged operations? There are two ways: the direct way is to make a "hypercall", to make requests of the Host Itself. [include/asm-x86/lguest_hcall.h]

- Our hypercall mechanism uses the highest unused trap code (traps 32 and above are used by real hardware interrupts).

```
static inline unsigned long hcall(unsigned long call,  unsigned long arg1,
                                  unsigned long arg2, unsigned long arg3)
{
        asm volatile("int $" __stringify(LGUEST_TRAP_ENTRY)
                     /* The call in %eax (aka "a") might be overwritten */
                     : "=a"(call)
                       /* The arguments are in %eax, %edx, %ebx & %ecx */
                     : "a"(call), "d"(arg1), "b"(arg2), "c"(arg3)
                       /* "memory" means this might write somewhere in memory.
                        * This isn't true for all calls, but it's safe to tell
                        * gcc that it might happen so it doesn't get clever. */
                     : "memory");
        return call;
}
```

replace example with hcall: lguest_set_pte_at()

```
static void lguest_set_pte_at(struct mm_struct *mm, unsigned long addr,
                              pte_t *ptep, pte_t pteval)
{
        *ptep = pteval;
        lazy_hcall(LHCALL_SET_PTE, __pa(mm->pgd), addr, pteval.pte_low);
}
```

lguest_data and a replace example: save_fl()

The **second method of communicating with the Host** is to via "struct lguest_data". [include/linux/lguest.h]

Once the Guest's initialization hypercall tells the Host where this is, the Guest and Host **both publish information in it**.

Example: we keep an "irq_enabled" field inside our "struct lguest_data", which the Guest can update with a single instruction. The Host knows to check there before it tries to deliver an interrupt.

/* **save_flags()** is expected to return the processor state (ie. "flags"). The
 * flags word contains all kind of stuff, but in practice **Linux only cares**
 * **about the interrupt flag**. Our "save_flags()" just returns that. */

```
static unsigned long save_fl(void)
{
return lguest_data.irq_enabled;
}
```

lguest_init(): start kernel !

.....
.....

/* Now we're set up, call i386_start_kernel() in head32.c and **we proceed**
 *** to boot as normal**. It never returns. */
i386_start_kernel();
}

host

- initialization
- run guest
- hypercalls
- shadow paging
- etc

module initialization

Let's begin at the initialization routine for the Host's lg module.

```
static int __init init(void)
{
	/* First we put the Switcher up in very high virtual memory. */
	err = map_switcher();

	/* Now we set up the pagetable implementation for the Guests. */
	err = init_pagetables(switcher_page, SHARED_SWITCHER_PAGES);

	/* We might need to reserve an interrupt vector. */
	err = init_interrupts();

	/* /dev/lguest needs to be registered. */
	err = lguest_device_init();

	/* Finally we do some architecture-specific setup. */
	lguest_arch_host_init();

}
```

map_switcher()

The Switcher code must be at the same virtual address in the Guest as the Host since it will be running as the switchover occurs.

Trying to map memory at a particular address is an unusual thing to do, so it's not a simple one-liner.

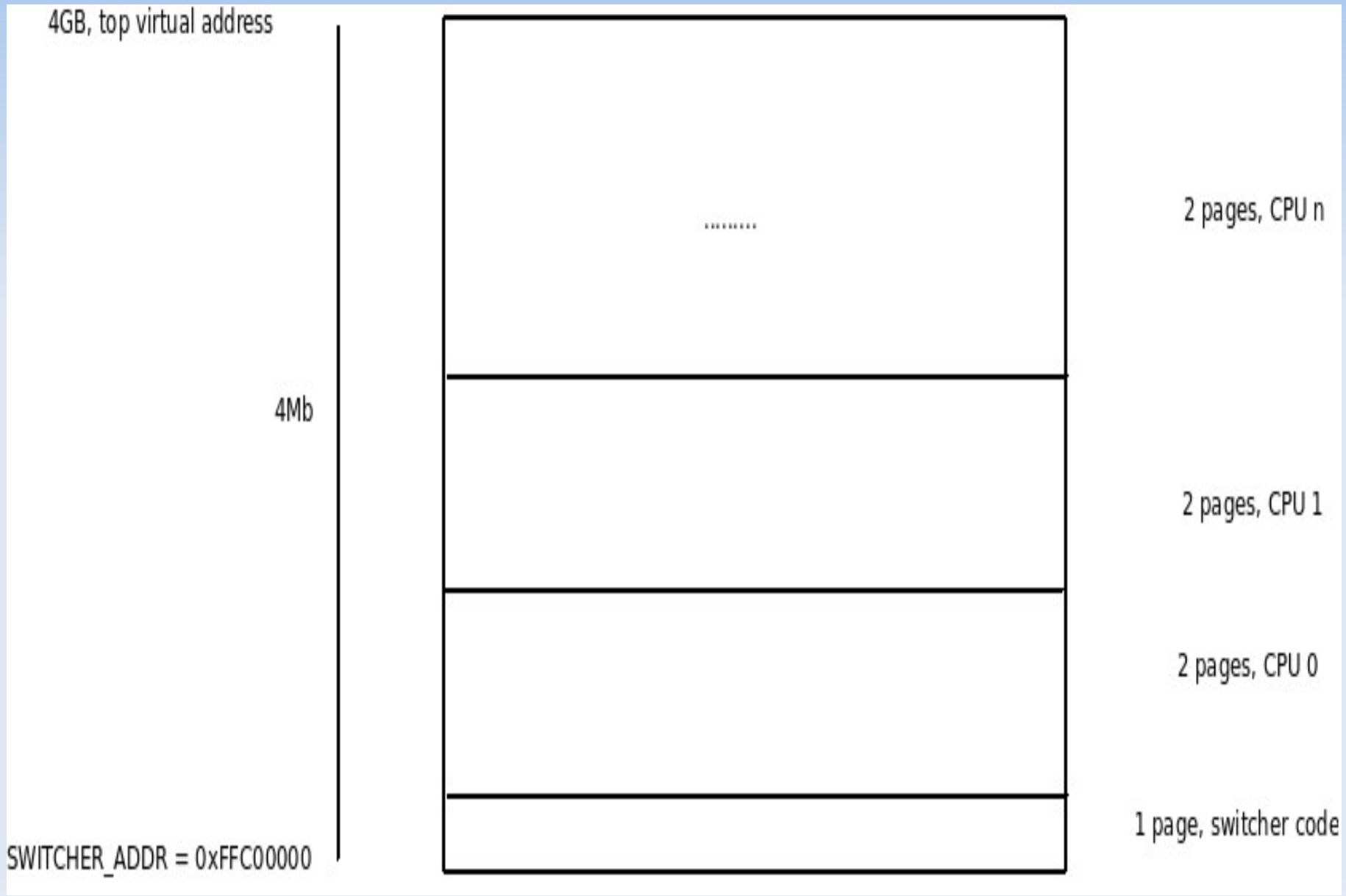
```
static __init int map_switcher(void)
{
    /* Now we actually allocate the pages. The Guest will see these pages,
         * so we make sure they're zeroed. */
       for (i = 0; i < TOTAL_SWITCHER_PAGES; i++) {
           unsigned long addr = get_zeroed_page(GFP_KERNEL);
           switcher_page[i] = virt_to_page(addr);
       }
```

map_switcher()

Now we reserve the "virtual memory area" we want: **0xFFC00000**
SWITCHER_ADDR

```
switcher_vma = __get_vm_area(TOTAL_SWITCHER_PAGES * PAGE_SIZE,
                    VM_ALLOC, SWITCHER_ADDR, SWITCHER_ADDR
                    + (TOTAL_SWITCHER_PAGES+1) * PAGE_SIZE);

    /* This code actually sets up the pages we've allocated to appear at
     * SWITCHER_ADDR....  */
    pagep = switcher_page;
    err = map_vm_area(switcher_vma, PAGE_KERNEL, &pagep);

    /* Now the Switcher is mapped at the right address, we can't fail!
     * Copy in the compiled-in Switcher code (from <arch>_switcher.S). */
    memcpy(switcher_vma->addr, start_switcher_text,
           end_switcher_text - start_switcher_text);
```

switcher mapped in high address



init_pagetables()

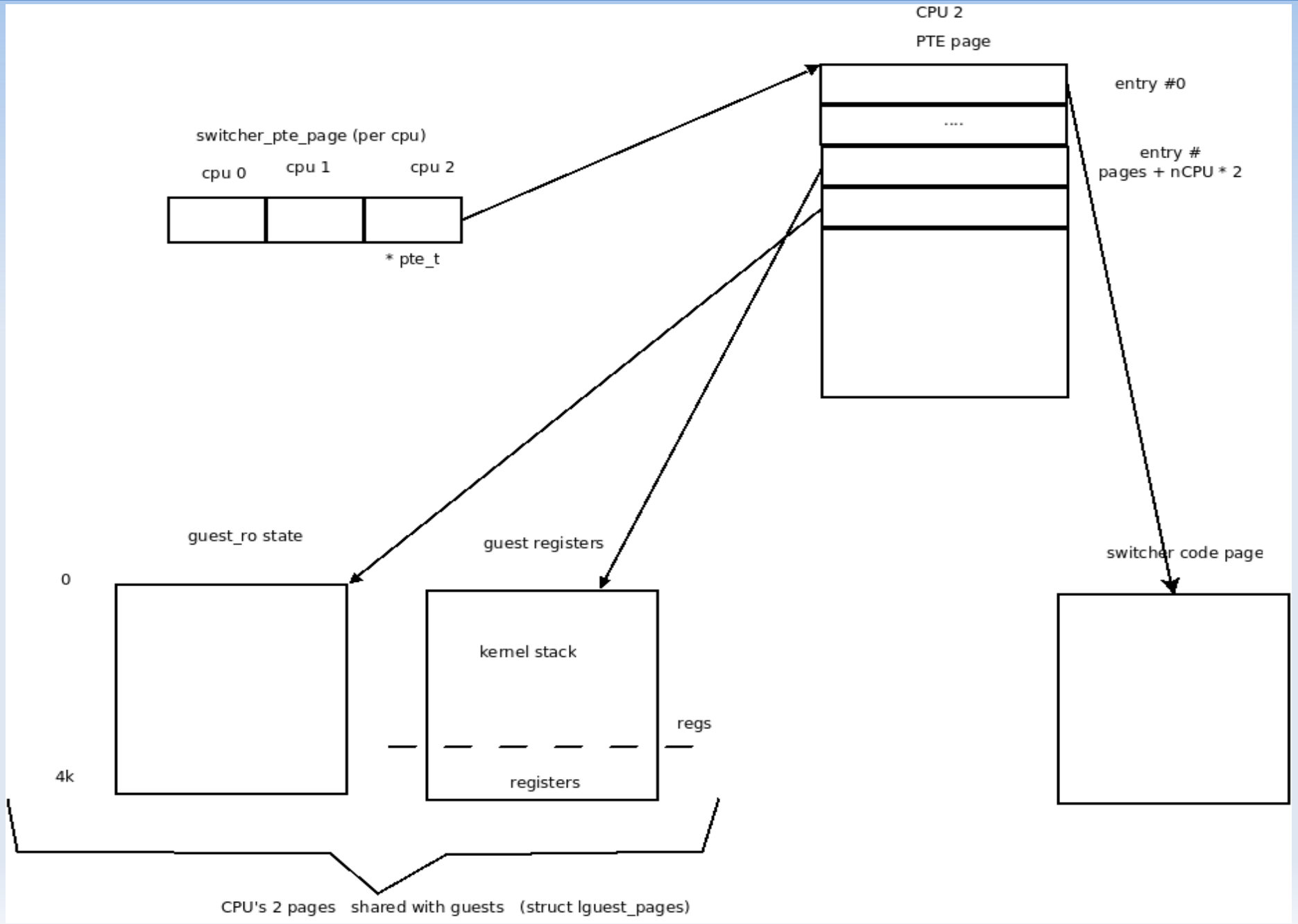
At boot or module load time, init_pagetables() allocates and populates the Switcher PTE page for each CPU.

```
__init int init_pagetables(struct page **switcher_page, unsigned int pages)
{
        unsigned int i;

        for_each_possible_cpu(i) {
                switcher_pte_page(i) = (pte_t *)get_zeroed_page(GFP_KERNEL);
                if (!switcher_pte_page(i)) {
                        free_switcher_pte_pages();
                        return -ENOMEM;
                }
                populate_switcher_pte_page(i, switcher_page, pages);
        }
        return 0;
}
```

populate_switcher_pte_page()

```
static __init void populate_switcher_pte_page(unsigned int cpu,
                                              struct page *switcher_page[], unsigned int pages)
{
        unsigned int i;
        pte_t *pte = switcher_pte_page(cpu);

        /* The first entries are easy: they map the Switcher code. */
        for (i = 0; i < pages; i++) {
              pte[i] = mk_pte(switcher_page[i], __pgprot(_PAGE_PRESENT|_PAGE_ACCESSED));
        }

        /* The only other thing we map is this CPU's pair of pages. */
        i = pages + cpu*2;

        /* First page (Guest registers) is writable from the Guest */
        pte[i] = pfn_pte(page_to_pfn(switcher_page[i]),
                         __pgprot(_PAGE_PRESENT|_PAGE_ACCESSED|_PAGE_RW));

        /* The second page contains the "struct lguest_ro_state", and is
         * read-only. */
        pte[i+1] = pfn_pte(page_to_pfn(switcher_page[i+1]),
                           __pgprot(_PAGE_PRESENT|_PAGE_ACCESSED));
}
```

smp switcher pages



lguest_arch_host_init()

Set up the Switcher's per-cpu areas. **Each CPU gets two pages of its own within the high-mapped region (aka. "struct lguest_pages").** Much of this can be initialized now, but **some depends on what Guest we are running (which is set up in copy_in_guest_info()).**

```
for_each_possible_cpu(i) {
    /* lguest_pages() returns this CPU's two pages. */
    struct lguest_pages *pages = lguest_pages(i);
    struct lguest_ro_state *state = &pages->state;

    /* The Global Descriptor Table: the Host has a different one
     * for each CPU. We keep a descriptor for the GDT which says
     * where it is and how big it is */
    state->host_gdt_desc.size = GDT_SIZE-1;
    state->host_gdt_desc.address = (long)get_cpu_gdt_table(i);

    /* All CPUs on the Host use the same Interrupt Descriptor
     * Table, so we just use store_idt(), which gets this CPU's IDT
     * descriptor. */
    store_idt(&state->host_idt_desc);
```

lguest_arch_host_init()

```
/* The descriptors for the Guest's GDT and IDT can be filled out now, too. */
        state->guest_idt_desc.size = sizeof(state->guest_idt)-1;
        state->guest_idt_desc.address = (long)&state->guest_idt;
        state->guest_gdt_desc.size = sizeof(state->guest_gdt)-1;
        state->guest_gdt_desc.address = (long)&state->guest_gdt;

 /* We know where we want the stack to be when the Guest enters the
Switcher: in pages->regs.  The stack grows upwards, so we start it at the end
of that structure. */
        state->guest_tss.sp0 = (long)(&pages->regs + 1);
        state->guest_tss.ss0 = LGUEST_DS;

/* x86 can have a finegrained bitmap which indicates what I/O ports the
process can use.  We set it to the end of our structure, meaning "none". */
        state->guest_tss.io_bitmap_base = sizeof(state->guest_tss);

 /* Some GDT entries are the same across all Guests, so we can set them up
now. */
        setup_default_gdt_entries(state);
 /* Most IDT entries are the same for all Guests, too.*/
        setup_default_idt_entries(state, default_idt_entries);
```

lguest_arch_host_init()

```
/* The Host needs to be able to use the LGUEST segments on this CPU */
        get_cpu_gdt_table(i)[GDT_ENTRY_LGUEST_CS] = FULL_EXEC_SEGMENT;
        get_cpu_gdt_table(i)[GDT_ENTRY_LGUEST_DS] = FULL_SEGMENT;
      }  /* for each cpu */
```

```
/* In the Switcher, we want the %cs segment register to use the
     * LGUEST_CS GDT entry: we've put that in the Host and Guest GDTs, so
     * it will be undisturbed when we switch.  To change %cs and jump we
     * need this structure to feed to Intel's "lcall" instruction. */
    lguest_entry.offset = (long)switch_to_guest + switcher_offset();
    lguest_entry.segment = LGUEST_CS;
```

setup_default_gdt_entries()

```
void setup_default_gdt_entries(struct lguest_ro_state *state)
{
        struct desc_struct *gdt = state->guest_gdt;
        unsigned long tss = (unsigned long)&state->guest_tss;

        /* The Switcher segments are full 0-4G segments, privilege level 0 */
        gdt[GDT_ENTRY_LGUEST_CS] = FULL_EXEC_SEGMENT;
        gdt[GDT_ENTRY_LGUEST_DS] = FULL_SEGMENT;

        /* The TSS segment refers to the TSS entry for this particular CPU.
         * Forgive the magic flags: the 0x8900 means the entry is Present, it's
         * privilege level 0 Available 386 TSS system segment, and the 0x67
         * means Saturn is eclipsed by Mercury in the twelfth house. */
        gdt[GDT_ENTRY_TSS].a = 0x00000067 | (tss << 16);
        gdt[GDT_ENTRY_TSS].b = 0x00008900 | (tss & 0xFF000000)
              | ((tss >> 16) & 0x000000FF);
}
```

setup_default_idt_entries()

```
/* When the Guest first starts, we put default entries into the IDT. */
   void setup_default_idt_entries(struct lguest_ro_state *state, const
   unsigned long *def)
   {
       unsigned int i;

       for (i = 0; i < ARRAY_SIZE(state->guest_idt); i++)
           default_idt_entry(&state->guest_idt[i], i, def[i], NULL);
   }
```

The default entry for each interrupt points into the Switcher routines which simply return to the Host. The run_guest() loop will then call deliver_trap() to bounce it back into the Guest.

default_idt_entry

```
static void default_idt_entry(struct desc_struct *idt, int trap,
                const unsigned long handler, const struct desc_struct *base)
{
    /* A present interrupt gate. */
    u32 flags = 0x8e00;

    /* Set the privilege level on the entry for the hypercall: this allows
     * the Guest to use the "int" instruction to trigger it. */
    if (trap == LGUEST_TRAP_ENTRY)
        flags |= (GUEST_PL << 13);
    else if (base)
        /* Copy priv. level from what Guest asked for.  This allows
         * debug (int 3) traps from Guest userspace, for example. */
        flags |= (base->b & 0x6000);

    /* Now pack it into the IDT entry in its weird format. */
    idt->a = (LGUEST_CS<<16) | (handler&0x0000FFFF);
    idt->b = (handler&0xFFFF0000) | flags;
}
```

host's run_guest () loop

```
/* We stop running once the Guest is dead. */
   while (!cpu->lg->dead) {
       /* First we run any hypercalls the Guest wants done. */
       if (cpu->hcall)
           do_hypercalls(cpu);

 /* It's possible the Guest did a NOTIFY hypercall to the Launcher, in which
case we return from the read() now. */
       if (cpu->pending_notify) {
           if (put_user(cpu->pending_notify, user))
               return -EFAULT;
           return sizeof(cpu->pending_notify);
       }

    /* If Waker set break_out, return to Launcher. */
       if (cpu->break_out)
           return -EAGAIN;

 /* Check if there are any interrupts which can be delivered now:if so, this
sets up the hander to be executed when we next run the Guest. */
       maybe_do_interrupt(cpu);
```

host's run_guest () loop

```
/* If the Guest asked to be stopped, we sleep.  The Guest's clock timer or
LHCALL_BREAK from the Waker will wake us. */
        if (cpu->halted) {
            set_current_state(TASK_INTERRUPTIBLE);
            schedule();
            continue;
        }

/* OK, now we're ready to jump into the Guest.  First we put up he "Do Not
Disturb" sign: */
        local_irq_disable();

        /* Actually run the Guest until something happens. */
        lguest_arch_run_guest(cpu);

        /* Now we're ready to be interrupted or moved to other CPUs */
        local_irq_enable();

        /* Now we deal with whatever happened to the Guest. */
        lguest_arch_handle_trap(cpu);
    }
```

void lguest_arch_run_guest()

.....

/* Now we actually run the Guest. It will return when something interesting happens */
run_guest_once(cpu, lguest_pages(raw_smp_processor_id()));

.....

/* If the Guest page faulted, then the cr2 register will tell us the bad virtual address. We have to grab this now, because **once we re-enable interrupts an interrupt could fault and thus overwrite cr2**, or we could even move off to a different CPU. */

```
if (cpu->regs->trapnum == 14)
        cpu->arch.last_pagefault = read_cr2();
```

.....

}

lguest_arch_handle_trap() switch

switch (cpu->regs->trapnum) {

case 13: /* We've intercepted a General Protection Fault. */
/* Check if this was one of those annoying IN or OUT instructions which we need to emulate. If so, we just go back into the Guest after we've done it. */
if (cpu->regs->errcode == 0)
if (**emulate_insn(cpu)**) return;
break;

case 14: /* We've intercepted a Page Fault. */
/* The Guest accessed a virtual address that wasn't mapped. This happens a lot: we don't actually set up most of the page tables for the Guest at all when we start: as it runs it asks for more and more, and we set them up as required. In this case, we don't even tell the Guest that the fault happened. /* The errcode tells whether this was a read or a write, and whether kernel or userspace code. */
if (**demand_page(cpu, cpu->arch.last_pagefault, cpu->regs->errcode)**)
return;

/* OK, it's really not there (or not OK): **the Guest needs to know**. We write out the cr2 value so it knows where the fault occurred. */
if (cpu->lg->lguest_data && put_user(cpu->arch.last_pagefault, &cpu->lg->lguest_data->cr2))
kill_guest(cpu, "Writing cr2");
break;

lguest_arch_handle_trap()

```
    case 32 ... 255:
        /* These values mean a real interrupt occurred, in which case the Host handler
has already been run.  We just do a friendly check if another process should now
be run, then return to run the Guest again */
        cond_resched();
        return;


    case LGUEST_TRAP_ENTRY:
        /* Our 'struct hcall_args' maps directly over our regs: we set up the pointer now
to indicate a hypercall is pending. */
        cpu->hcall = (struct hcall_args *)cpu->regs;
        return;
    }
```

lguest_arch_handle_trap()

```
    /* We didn't handle the trap, so it needs to go to the Guest. */
    if (!deliver_trap(cpu, cpu->regs->trapnum))

 /* If the Guest doesn't have a handler (either it hasn't registered any yet, or it's one
of the faults we don't let it handle), it dies with this cryptic error message. */
        kill_guest(cpu, "unhandled trap %li at %#lx (%#lx)",
                   cpu->regs->trapnum, cpu->regs->eip,
                   cpu->regs->trapnum == 14 ? cpu->arch.last_pagefault
                   : cpu->regs->errcode);
}
```

deliver_trap()

```
/* deliver_trap() returns true if it could deliver the trap. */
   int deliver_trap(struct lg_cpu *cpu, unsigned int num)
   {
      /* Trap numbers are always 8 bit, but we set an impossible trap number
       * for traps inside the Switcher, so check that here. */
      if (num >= ARRAY_SIZE(cpu->arch.idt))
          return 0;

      /* Early on the Guest hasn't set the IDT entries (or maybe it put a
       * bogus one in): if we fail here, the Guest will be killed. */
      if (!idt_present(cpu->arch.idt[num].a, cpu->arch.idt[num].b))
                return 0;

      set_guest_interrupt(cpu, cpu->arch.idt[num].a,
                          cpu->arch.idt[num].b, has_err(num));
      return 1;
   }
```

do_hcall()

/* This is the core hypercall routine: where the Guest gets what it wants. */

```
static void do_hcall(struct lg_cpu *cpu, struct hcall_args *args)
{
    switch (args->arg0){
    case LHCALL_FLUSH_ASYNC:
        /* This call does nothing, except by breaking out of the Guest
         * it makes us process all the asynchronous hypercalls. */
        break;

 case LHCALL_LGUEST_INIT:
        /* You can't get here unless you're already initialized.  Don't
         * do that. */
        kill_guest(cpu, "already have lguest_data");
        break;
....
```

do_hcall()

```
case LHCALL_FLUSH_TLB:
    /* FLUSH_TLB comes in two flavors, depending on the argument: */
    if (args->arg1)
        guest_pagetable_clear_all(cpu);
    else
        guest_pagetable_flush_user(cpu);
    break;

case LHCALL_NEW_PGTABLE:
    guest_new_pagetable(cpu, args->arg1);
    break;

case LHCALL_SET_STACK:
    guest_set_stack(cpu, args->arg1, args->arg2, args->arg3);
    break;

case LHCALL_SET_PTE:
    guest_set_pte(cpu, args->arg1, args->arg2, __pte(args->arg3));
    break;
```

maybe_do_interrupt()

```
/* Take our "irqs_pending" array and remove any interrupts the Guest wants blocked: the
   result ends up in "blk". */
if (copy_from_user(&blk, cpu->lg->lguest_data->blocked_interrupts, sizeof(blk)))      return;
bitmap_andnot(blk, cpu->irqs_pending, blk, LGUEST_IRQS);

/* Find the first interrupt. */
irq = find_first_bit(blk, LGUEST_IRQS);
/* None?  Nothing to do */
if (irq >= LGUEST_IRQS)
          return;
if (cpu->halted) {        /* If they're halted, interrupts restart them. */
        /* Re-enable interrupts. */
        if (put_user(X86_EFLAGS_IF, &cpu->lg->lguest_data->irq_enabled))
                kill_guest(cpu, "Re-enabling interrupts");
        cpu->halted = 0;
} else {
      /* Otherwise we check if they have interrupts disabled. */
      u32 irq_enabled;
      if (get_user(irq_enabled, &cpu->lg->lguest_data->irq_enabled))
           irq_enabled = 0;
      if (!irq_enabled)
            return;
}
```

maybe_do_interrupt ()

```
/* Look at the IDT entry the Guest gave us for this interrupt. The first 32
  (FIRST_EXTERNAL_VECTOR) entries are for traps, so we skip over them. */
   idt = &cpu->arch.idt[FIRST_EXTERNAL_VECTOR+irq];

   /* If they don't have a handler (yet?), we just ignore it */
      if (idt_present(idt->a, idt->b)) {

       /* OK, mark it no longer pending and deliver it. */
          clear_bit(irq, cpu->irqs_pending);

       /* set_guest_interrupt() takes the interrupt descriptor and a
            * flag to say whether this interrupt pushes an error code onto
            * the stack as well: virtual interrupts never do. */
          set_guest_interrupt(cpu, idt->a, idt->b, 0);
      }
```

set_guest_interrupt ()

There are **two cases for interrupts**: one where the **Guest is already in the kernel,** and a more complex one where the **Guest is in userspace**.
We check the privilege level to find out:

```
 if ((cpu->regs->ss&0x3) != GUEST_PL) {
        /* The Guest told us their kernel stack with the SET_STACK
         * hypercall: both the virtual address and the segment */
        virtstack = cpu->esp1;
        ss = cpu->ss1;
        origstack = gstack = guest_pa(cpu, virtstack);

    /* We push the old stack segment and pointer onto the new stack: when the
   Guest does an "iret" back from the interrupt handler the CPU will notice they're
   dropping privilege levels and expect these here. */
        push_guest_stack(cpu, &gstack, cpu->regs->ss);
        push_guest_stack(cpu, &gstack, cpu->regs->esp);
    } else {
        /* We're staying on the same Guest (kernel) stack. */
        virtstack = cpu->regs->esp;
        ss = cpu->regs->ss;
        origstack = gstack = guest_pa(cpu, virtstack);
   }
```

set_guest_interrupt ()

Remember that **we never let the Guest actually disable interrupts**, so the "Interrupt Flag" bit is always set. We **copy that bit from the Guest's "irq_enabled" field into the eflags word: we saw the Guest copy it back in "lguest_iret".**

```
    eflags = cpu->regs->eflags;
   if (get_user(irq_enable, &cpu->lg->lguest_data->irq_enabled) == 0
                                                    && !(irq_enable & X86_EFLAGS_IF))
        eflags &= ~X86_EFLAGS_IF;

  /* An interrupt is expected to push three things on the stack: the old "eflags" word,
       the old code segment, and the old instruction pointer. */
     push_guest_stack(cpu, &gstack, eflags);
     push_guest_stack(cpu, &gstack, cpu->regs->cs);
     push_guest_stack(cpu, &gstack, cpu->regs->eip);
     if (has_err)  push_guest_stack(cpu, &gstack, cpu->regs->errcode);

     cpu->regs->ss = ss;
     cpu->regs->esp = virtstack + (gstack - origstack);
     cpu->regs->cs = (__KERNEL_CS|GUEST_PL);
     cpu->regs->eip = idt_address(lo, hi);

  /* There are two kinds of interrupt handlers: 0xE is an "interrupt gate" which expects
  interrupts to be disabled on entry. */
     if (idt_type(lo, hi) == 0xE)
         if (put_user(0, &cpu->lg->lguest_data->irq_enabled))
             kill_guest(cpu, "Disabling interrupts");
```

load_guest_idt_entry()

While we're here, dealing with delivering traps and interrupts to the Guest, we might as well complete the picture: how the Guest tells us where it wants them to go. This would be simple, except **making traps fast requires some tricks.**

```
/*We saw the Guest setting Interrupt Descriptor Table (IDT) entries with the
LHCALL_LOAD_IDT_ENTRY hypercall before: that comes here. */
void load_guest_idt_entry(struct lg_cpu *cpu, unsigned int num, u32 lo, u32 hi)
{
    /* Guest never handles: NMI, doublefault, spurious interrupt or
     * hypercall.  We ignore when it tries to set them. */
    if (num == 2 || num == 8 || num == 15 || num == LGUEST_TRAP_ENTRY)
        return;

/* Mark the IDT as changed: next time the Guest runs we'll know we have to copy this again.*/
    cpu->changed |= CHANGED_IDT;

    /* Check that the Guest doesn't try to step outside the bounds. */
    if (num >= ARRAY_SIZE(cpu->arch.idt))
        kill_guest(cpu, "Setting idt entry %u", num);
    else
        set_trap(cpu, &cpu->arch.idt[num], num, lo, hi);
}
```

set_trap ()

This is the routine which actually checks the Guest's IDT entry and transfers it into the entry in "struct lguest":

```
static void set_trap(struct lg_cpu *cpu, struct desc_struct *trap,
                     unsigned int num, u32 lo, u32 hi)
{
    u8 type = idt_type(lo, hi);

    /* We only support interrupt and trap gates. */
    if (type != 0xE && type != 0xF)
        kill_guest(cpu, "bad IDT type %i", type);

/* We only copy the handler address, present bit, privilege level and type.
The privilege level controls where the trap can be triggered manually with an
"int" instruction.  This is usually GUEST_PL, except for system calls which
userspace can use. */
    trap->a = ((__KERNEL_CS|GUEST_PL)<<16) | (lo&0x0000FFFF);
    trap->b = (hi&0xFFFFEF00);
```

guest_set_stack()

Direct traps also mean that **we need to know whenever the Guest wants to use a different kernel stack, so we can change the IDT entries to use that stack**.

In Linux each process has its own kernel stack, so this happens a lot: we change stacks on each context switch. LHCALL_SET_STACK...*/

void guest_set_stack(struct lg_cpu *cpu, u32 seg, u32 esp, unsigned int pages)

```
{
..... (privilege controls and count of pages <=2)....

    /* Save where the stack is, and how many pages */
    cpu->ss1 = seg;
    cpu->esp1 = esp;
    cpu->lg->stack_pages = pages;

    /* Make sure the new stack pages are mapped */
    pin_stack_pages(cpu);
}
```

The Page Tables Code

We use two-level page tables for the Guest.

The Guest keeps page tables, but we maintain the actual ones here: these are called **"shadow" page tables**. Which is a very Guest-centric name: **these are the real page tables the CPU uses**, although we keep them up to date to reflect the Guest's.

There are seven parts to this:

* **(i) Looking up a page table entry when the Guest faults,**
* (ii) Making sure the Guest stack is mapped,
* **(iii) Setting up a page table entry when the Guest tells us one has changed,**
* **(iv) Switching page tables,**
* (v) Flushing (throwing away) page tables,
* **(vi) Mapping the Switcher when the Guest is about to run,**
* **(vii) Setting up the page tables initially.**

Note that all this complexity in juggling shadow page tables in sync with the Guest's page tables is for one reason: **for most Guests this page table dance determines how bad performance will be**. This is why Xen uses exotic direct Guest pagetable manipulation, and why both Intel and AMD have implemented shadow page table support directly into hardware.

helper functions

There are two functions which return pointers to the shadow (aka "real") page tables.

spgd_addr() takes the virtual address and returns a pointer to the top-level page directory entry (PGD) for that address. Since we keep track of several page tables, the "i" argument tells us which one we're interested in (it's usually the current one). */

```
static pgd_t *spgd_addr(struct lg_cpu *cpu, u32 i, unsigned long vaddr)
{
        unsigned int index = pgd_index(vaddr);
      .........
        /* Return a pointer index'th pgd entry for the i'th page table. */
        return &cpu->lg->pgdirs[i].pgdir[index];
}
```

This routine then takes the page directory entry returned above, which contains the address of the page table entry (PTE) page. It then returns a pointer to the PTE entry for the given address

```
static pte_t *spte_addr(pgd_t spgd, unsigned long vaddr)
{
        pte_t *page = __va(pgd_pfn(spgd) << PAGE_SHIFT);
        ..............
        return &page[(vaddr >> PAGE_SHIFT) % PTRS_PER_PTE];
}
```

helper functions

These two functions just like the above two, except they access the Guest page tables. Hence they return a Guest address.

```
static unsigned long gpgd_addr(struct lg_cpu *cpu, unsigned long vaddr)
{
        unsigned int index = vaddr >> (PGDIR_SHIFT);
        return cpu->lg->pgdirs[cpu->cpu_pgd].gpgdir + index * sizeof(pgd_t);
}
```

```
static unsigned long gpte_addr(pgd_t gpgd, unsigned long vaddr)
{
        unsigned long gpage = pgd_pfn(gpgd) << PAGE_SHIFT;
        .......
        return gpage + ((vaddr>>PAGE_SHIFT) % PTRS_PER_PTE) *
   sizeof(pte_t);
}
```

Looking up a page table when the Guest faults

```
 * We saw this call in run_guest(): when we see a page fault in the Guest, we
 * come here.  That's because we only set up the shadow page tables lazily as
 * they're needed, so we get page faults all the time and quietly fix them up
 * and return to the Guest without it knowing.
 *
 * If we fixed up the fault (ie. we mapped the address), this routine returns
 * true.  Otherwise, it was a real fault and we need to tell the Guest.

int demand_page(struct lg_cpu *cpu, unsigned long vaddr, int errcode)
{
        pgd_t gpgd,  *spgd;
        unsigned long gpte_ptr;
        pte_t gpte,  *spte;

        /* First step: get the top-level Guest page table entry. */
        gpgd = lgread(cpu, gpgd_addr(cpu, vaddr), pgd_t);

        /* Toplevel not present?  We can't map it in. */
        if (!(pgd_flags(gpgd) & _PAGE_PRESENT))
              return 0;
```

demand_page()

```
/* Now look at the matching shadow entry. */
spgd = spgd_addr(cpu, cpu->cpu_pgd, vaddr);

if (!(pgd_flags(*spgd) & _PAGE_PRESENT)) {
        /* No shadow entry: allocate a new shadow PTE page. */
        unsigned long ptepage = get_zeroed_page(GFP_KERNEL);
        .........

        /* We check that the Guest pgd is OK. */
        check_gpgd(cpu, gpgd);

        /* And we copy the flags to the shadow PGD entry.  The page
         * number in the shadow PGD is the page we just allocated. */
        *spgd = __pgd(__pa(ptepage) | pgd_flags(gpgd));
}
```

demand_page()

```
/* OK, now we look at the lower level in the Guest page table: keep its
 * address, because we might update it later. */
gpte_ptr = gpte_addr(gpgd, vaddr);
gpte = lgread(cpu, gpte_ptr, pte_t);

/* If this page isn't in the Guest page tables, we can't page it in. */
if (!(pte_flags(gpte) & _PAGE_PRESENT))
	return 0;

/* Check they're not trying to write to a page the Guest wants
 * read-only (bit 2 of errcode == write). */
if ((errcode & 2) && !(pte_flags(gpte) & _PAGE_RW))
	return 0;

/* User access to a kernel-only page? (bit 3 == user access) */
if ((errcode & 4) && !(pte_flags(gpte) & _PAGE_USER))
	return 0;

/* Check that the Guest PTE flags are OK, and the page number is below
 * the pfn_limit (ie. not mapping the Launcher binary). */
check_gpte(cpu, gpte);
```

demand_page()

```
/* Add the _PAGE_ACCESSED and (for a write) _PAGE_DIRTY flag */
gpte = pte_mkyoung(gpte);
if (errcode & 2)
      gpte = pte_mkdirty(gpte);

/* Get the pointer to the shadow PTE entry we're going to set. */
spte = spte_addr(*spgd, vaddr);

.................

*spte = gpte_to_spte(cpu, gpte, 0);
```

demand_page()

```
        /* Finally, we write the Guest PTE entry back: we've set the
         * _PAGE_ACCESSED and maybe the _PAGE_DIRTY flags. */
        lgwrite(cpu, gpte_ptr, pte_t, gpte);

        /* The fault is fixed, the page table is populated, the mapping
         * manipulated, the result returned and the code complete.  A small
         * delay and a trace of alliteration are the only indications the Guest
         * has that a page fault occurred at all. */
        return 1;
}
```

gpte_to_spte()

Converting a Guest page table entry to a shadow (ie. real) page table entry:The flags are (almost) the same, but the Guest PTE contains a virtual page frame: the CPU needs the real page frame.

```
static pte_t gpte_to_spte(struct lg_cpu *cpu, pte_t gpte, int write)
{
/*The Guest sets the global flag, because it thinks that it is using PGE.  We only told it to use
   PGE so it would tell us whether it was flushing a kernel mapping or a userspace
   mapping.  We don't actually use the global bit, so throw it away. */
        flags = (pte_flags(gpte) & ~_PAGE_GLOBAL);

        /* The Guest's pages are offset inside the Launcher. */
        base = (unsigned long)cpu->lg->mem_base / PAGE_SIZE;
```

get_pfn() takes a page number given by the Guest and converts it to an actual, physical page number.

```
        pfn = get_pfn(base + pte_pfn(gpte), write);
        .........
        /* Now we assemble our shadow PTE from the page number and flags. */
        return pfn_pte(pfn, __pgprot(flags));
}
```

Setting up a PT entry when the Guest tells us : guest_set_pmd()

What happens when the Guest asks for a page table to be updated?

We already saw that **demand_page() will fill in the shadow page tables when needed**, so we can simply **remove shadow page table entries** whenever the Guest tells us they've changed. **When the Guest tries to use the new entry it will fault and demand_page() will fix it up**.

```
void guest_set_pmd(struct lguest *lg, unsigned long gpgdir, u32 idx)
{
        int pgdir;

        /* The kernel seems to try to initialize this early on: we ignore its
         * attempts to map over the Switcher. */
        if (idx >= SWITCHER_PGD_INDEX)
                return;

        /* If they're talking about a page table we have a shadow for... */
        pgdir = find_pgdir(lg, gpgdir);
        if (pgdir < ARRAY_SIZE(lg->pgdirs))
                release_pgd(lg, lg->pgdirs[pgdir].pgdir + idx);    /* ... throw it away. */
}
```

guest_set_pte()

We keep track of several different page tables (the **Guest uses one for each process, so it makes sense to cache at least a few**). Each of these have **identical kernel parts:** ie. every mapping above PAGE_OFFSET is the same for all processes. So when the page table above that address changes, we update all the page tables, not just the current one. This is rare. The benefit is that when we have to track a new page table, we can keep all the kernel mappings. This speeds up context switch immensely.

```
void guest_set_pte(struct lg_cpu *cpu, unsigned long gpgdir, unsigned long vaddr, pte_t gpte)
{
        /* Kernel mappings must be changed on all top levels.  Slow, but doesn't
         * happen often. */
        if (vaddr >= cpu->lg->kernel_address) {
                unsigned int i;
                for (i = 0; i < ARRAY_SIZE(cpu->lg->pgdirs); i++)
                        if (cpu->lg->pgdirs[i].pgdir)
                                do_set_pte(cpu, i, vaddr, gpte);
        } else {
                /* Is this page table one we have a shadow for? */
                int pgdir = find_pgdir(cpu->lg, gpgdir);
                if (pgdir != ARRAY_SIZE(cpu->lg->pgdirs))
                        /* If so, do the update. */
                        do_set_pte(cpu, pgdir, vaddr, gpte);
        }
}
```

do_set_pte()

This is the routine which actually sets the page table entry for then "idx"'th shadow page table.

```
static void do_set_pte(struct lg_cpu *cpu, int idx,  unsigned long vaddr, pte_t gpte)
{
	/* Look up the matching shadow page directory entry. */
	pgd_t *spgd = spgd_addr(cpu, idx, vaddr);

	/* If the top level isn't present, there's no entry to update. */
	if (pgd_flags(*spgd) & _PAGE_PRESENT) {
		/* Otherwise, we start by releasing the existing entry. */
		pte_t *spte = spte_addr(*spgd, vaddr);
		release_pte(*spte);

		/* If they're setting this entry as dirty or accessed, we might
		 * as well put that entry they've given us in now.  This shaves
		 * 10% off a copy-on-write micro-benchmark. */
		if (pte_flags(gpte) & (_PAGE_DIRTY | _PAGE_ACCESSED)) {
			check_gpte(cpu, gpte);
			*spte = gpte_to_spte(cpu, gpte, pte_flags(gpte) & _PAGE_DIRTY);
		} else
			/* Otherwise kill it and we can demand_page() it in later. */
			*spte = __pte(0);
	}
}
```

switching page tables

Guest changes page tables (LHCALL_NEW_PGTABLE). This occurs on almost every context switch.

```
void guest_new_pagetable(struct lg_cpu *cpu, unsigned long pgtable)
{
	int newpgdir, repin = 0;

	/* Look to see if we have this one already. */
	newpgdir = find_pgdir(cpu->lg, pgtable);

	/* If not, we allocate or mug an existing one: if it's a fresh one,
	 * repin gets set to 1. */
	if (newpgdir == ARRAY_SIZE(cpu->lg->pgdirs))
		newpgdir = new_pgdir(cpu, pgtable, &repin);

	/* Change the current pgd index to the new one. */
	cpu->cpu_pgd = newpgdir;

	/* If it was completely blank, we map in the Guest kernel stack */
	if (repin)
		pin_stack_pages(cpu);
}
```

new_pgdir()

```
/* And this is us, creating the new page directory.  If we really do allocate a new one (and so
   the kernel parts are not there), we set blank_pgdir. */
static unsigned int new_pgdir(struct lg_cpu *cpu, unsigned long gpgdir, int *blank_pgdir)
{
        unsigned int next;

        next = random32() % ARRAY_SIZE(cpu->lg->pgdirs);
        /* If it's never been allocated at all before, try now. */
        if (!cpu->lg->pgdirs[next].pgdir) {
                cpu->lg->pgdirs[next].pgdir =  (pgd_t *)get_zeroed_page(GFP_KERNEL);
                /* If the allocation fails, just keep using the one we have */
                if (!cpu->lg->pgdirs[next].pgdir)
                        next = cpu->cpu_pgd;
                else
                        /* This is a blank page, so there are no kernel
                         * mappings: caller must map the stack! */
                        *blank_pgdir = 1;
        }
        /* Record which Guest toplevel this shadows. */
        cpu->lg->pgdirs[next].gpgdir = gpgdir;
        /* Release all the non-kernel mappings. */
        flush_user_mappings(cpu->lg, next);
        return next;
}
```

segments

GDT entries are passed around as "**struct desc_struct**"s, which like IDT entries are split into two 32-bit members, "a" and "b". One day, someone will clean that up, and be declared a Hero. (No pressure, I'm just saying).

Anyway, the GDT entry contains a base (the start address of the segment), a limit (the size of the segment - 1), and some flags. Sounds simple, and it would be, except those zany Intel engineers decided that it was too boring to put the base at one end, the limit at the other, and the flags in between. They decided to shotgun the bits at random throughout the 8 bytes.

Like the IDT, we never simply use the GDT the Guest gives us. We keep a GDT for each CPU, and copy across the Guest's entries each time we want to run the Guest on that CPU.

load_guest_gdt()

```
/* This is where the Guest asks us to load a new GDT (LHCALL_LOAD_GDT).
 * We copy it from the Guest and tweak the entries. */

void load_guest_gdt(struct lg_cpu *cpu, unsigned long table, u32 num)
{
        /* We assume the Guest has the same number of GDT entries as the
         * Host, otherwise we'd have to dynamically allocate the Guest GDT. */
        if (num > ARRAY_SIZE(cpu->arch.gdt))
                kill_guest(cpu, "too many gdt entries %i", num);

        /* We read the whole thing in, then fix it up. */
        __lgread(cpu, cpu->arch.gdt, table, num * sizeof(cpu->arch.gdt[0]));
        fixup_gdt_table(cpu, 0, ARRAY_SIZE(cpu->arch.gdt));

        /* Mark that the GDT changed so the core knows it has to copy it again,
         * even if the Guest is run on the same CPU. */
        cpu->changed |= CHANGED_GDT;
}
```

guest_load_tls ()

```
* This is the fast-track version for just changing the three TLS entries.
 * Remember that this happens on every context switch, so it's worth
 * optimizing.  But wouldn't it be neater to have a single hypercall to cover
 * both cases? */  case LHCALL_LOAD_TLS:

void guest_load_tls(struct lg_cpu *cpu, unsigned long gtls)
{
        struct desc_struct *tls = &cpu->arch.gdt[GDT_ENTRY_TLS_MIN];

        __lgread(cpu, tls, gtls, sizeof(*tls)*GDT_ENTRY_TLS_ENTRIES);
        fixup_gdt_table(cpu, GDT_ENTRY_TLS_MIN, GDT_ENTRY_TLS_MAX+1);

        /* Note that just the TLS entries have changed. */
        cpu->changed |= CHANGED_GDT_TLS;
}

/* Once the Guest gave us new GDT entries, we fix them up a little.  We
 * don't care if they're invalid: the worst that can happen is a General
 * Protection Fault in the Switcher when it restores a Guest segment register
 * which tries to use that entry.  Then we kill the Guest for causing such a
 * mess: the message will be "unhandled trap 256". */
```

fixup_gdt_table()

```
static void fixup_gdt_table(struct lg_cpu *cpu, unsigned start, unsigned end)
{
        unsigned int i;
        for (i = start; i < end; i++) {
                /* We never copy these ones to real GDT, so we don't care what
                 * they say */
                if (ignored_gdt(i))
                        continue;

                /* Segment descriptors contain a privilege level: the Guest is
                 * sometimes careless and leaves this as 0, even though it's
                 * running at privilege level 1.  If so, we fix it here. */
                if ((cpu->arch.gdt[i].b & 0x00006000) == 0)
                        cpu->arch.gdt[i].b |= (GUEST_PL << 13);

                /* Each descriptor has an "accessed" bit.  If we don't set it
                 * now, the CPU will try to set it when the Guest first loads
                 * that entry into a segment register.  But the GDT isn't
                 * writable by the Guest, so bad things can happen. */
                cpu->arch.gdt[i].b |= 0x00000100;
        }
}
```

ignored_gdt()

There are **several entries we don't let the Guest set**. The TSS entry is the
* "Task State Segment" which controls all kinds of delicate things. The
* LGUEST_CS and LGUEST_DS entries are reserved for the Switcher, and
the
* the Guest can't be trusted to deal with double faults. */

```
static int ignored_gdt(unsigned int num)
{
        return (num == GDT_ENTRY_TSS
             || num == GDT_ENTRY_LGUEST_CS
             || num == GDT_ENTRY_LGUEST_DS
             || num == GDT_ENTRY_DOUBLEFAULT_TSS);
}
```

getting closer to the switcher

Remember that each CPU has two pages which are visible to the Guest when it runs on that CPU.

This has to contain the state for that Guest: we copy the state in just before we run the Guest.

Each Guest has "changed" flags which indicate what has changed in the Guest since it last ran. We saw this set in interrupts_and_traps.c and segments.c.

run_guest_once()

/* Copy the **guest-specific information into this CPU's "struct lguest_pages"** */
copy_in_guest_info(cpu, pages);
.....
/* Now: we push the "eflags" register on the stack, then do an "lcall".
This is how we change from using the kernel code segment to using the dedicated lguest code segment, as well as jumping into the Switcher.
The lcall also pushes the old code segment (KERNEL_CS) onto the stack, then the address of this call.
This **stack layout happens to exactly match the stack layout created by an interrupt...** */

asm volatile("pushf; lcall *lguest_entry"
/* This is how we tell GCC that %eax ("a") and %ebx ("b")
* are changed by this routine. The "=" means output. */ clobber is a dummy value
: "=a"(clobber), "=b"(clobber)
/* %eax contains the pages pointer. ("0" refers to the 0-th argument above, ie "a").
* %ebx contains the physical address of the Guest's top-level page directory. */
: "0"(pages), "1"(__pa(cpu->lg->pgdirs[cpu->cpu_pgd].pgdir))
/* We tell gcc that all these registers could change,
* which means we don't have to save and restore them in the Switcher. */
: "memory", "%edx", "%ecx", "%edi", "%esi");
}

copy_in_guest_info()

```
static void copy_in_guest_info(struct lg_cpu *cpu, struct lguest_pages *pages)
{
/* Copying all this data can be quite expensive.  We usually run the same Guest we ran
   last time (and that Guest hasn't run anywhere else meanwhile).  If that's not the case,
   we pretend everything in the Guest has changed. */
        if (__get_cpu_var(last_cpu) != cpu || cpu->last_pages != pages) {
                __get_cpu_var(last_cpu) = cpu;
                cpu->last_pages = pages;
                cpu->changed = CHANGED_ALL;
        }

        /* These copies are pretty cheap, so we do them unconditionally: */
        /* Save the current Host top-level page directory. */
        pages->state.host_cr3 = __pa(current->mm->pgd);

        /* Set up the Guest's page tables to see this CPU's pages (and no
         * other CPU's pages). */
        map_switcher_in_guest(cpu, pages);

        /* Set up the two "TSS" members which tell the CPU what stack to use
         * for traps which do directly into the Guest (ie. traps at privilege level 1). */
        pages->state.guest_tss.sp1 = cpu->esp1;
        pages->state.guest_tss.ss1 = cpu->ss1;
```

copy_in_guest_info()

```
        /* Copy direct-to-Guest trap entries. */
        if (cpu->changed & CHANGED_IDT)
                copy_traps(cpu, pages->state.guest_idt, default_idt_entries);

        /* Copy all GDT entries which the Guest can change. */
        if (cpu->changed & CHANGED_GDT)
                copy_gdt(cpu, pages->state.guest_gdt);

        /* If only the TLS entries have changed, copy them. */
        else if (cpu->changed & CHANGED_GDT_TLS)
                copy_gdt_tls(cpu, pages->state.guest_gdt);

        /* Mark the Guest as unchanged for next time. */
        cpu->changed = 0;
}
```

Like the IDT, we never simply use the GDT the Guest gives us. We keep a GDT for each CPU, and copy across the Guest's entries each time we want to run the Guest on that CPU.

map_switcher_in_guest()

The Switcher and the two pages for this CPU need to be visible in the Guest (and not the pages for other CPUs). We have the appropriate PTE pages for each CPU already set up, we just need to hook them in now we know which Guest is about to run on this CPU.

```
void map_switcher_in_guest(struct lg_cpu *cpu, struct lguest_pages *pages)
{
        pte_t *switcher_pte_page = __get_cpu_var(switcher_pte_pages);
        pgd_t switcher_pgd;
        pte_t regs_pte;
        unsigned long pfn;

        /* Make the last PGD entry for this Guest point to the Switcher's PTE
         * page for this CPU (with appropriate flags). */
        switcher_pgd = __pgd(__pa(switcher_pte_page) | __PAGE_KERNEL);
        cpu->lg->pgdirs[cpu->cpu_pgd].pgdir[SWITCHER_PGD_INDEX] = switcher_pgd;


/*When we're running the Guest, we want the Guest's "regs" page to appear where the
   first Switcher page for this CPU is. */

pfn = __pa(cpu->regs_page) >> PAGE_SHIFT;
regs_pte = pfn_pte(pfn, __pgprot(__PAGE_KERNEL));
switcher_pte_page[(unsigned long)pages/PAGE_SIZE%PTRS_PER_PTE] = regs_pte;
```

copy_traps()

We don't use the IDT entries in the "struct lguest" directly, instead we copy them into the IDT which we've set up for Guests on this CPU, just before we run the Guest. This routine does that copy.

```
void copy_traps(const struct lg_cpu *cpu, struct desc_struct *idt, const unsigned long *def)
{
   /* We can simply copy the direct traps, otherwise we use the default
    * ones in the Switcher: they will return to the Host. */
    for (i = 0; i < ARRAY_SIZE(cpu->arch.idt); i++) {
        const struct desc_struct *gidt = &cpu->arch.idt[i];

        /* If no Guest can ever override this trap, leave it alone. */
        if (!direct_trap(i))
            continue;

        if (idt_type(gidt->a, gidt->b) == 0xF)
            idt[i] = *gidt;
        else
            default_idt_entry(&idt[i], i, def[i], gidt);
    }
}
```

direct_trap()

This routine indicates if a particular trap number could be delivered directly.

```
static int direct_trap(unsigned int num)
{
    /* Hardware interrupts don't go to the Guest at all (except system call). */
    if (num >= FIRST_EXTERNAL_VECTOR && !could_be_syscall(num))
        return 0;

    /* The Host needs to see page faults (for shadow paging and to save the
     * fault address), general protection faults (in/out emulation) and
     * device not available (TS handling), and of course, the hypercall
     * trap. */
    return num != 14 && num != 13 && num != 7 && num != LGUEST_TRAP_ENTRY;
}
```

copy_gdt()

```
/* When the Guest is run on a different CPU, or the GDT entries have changed,
   copy_gdt() is called to copy the Guest's GDT entries across to this CPU's GDT. */
void copy_gdt(const struct lg_cpu *cpu, struct desc_struct *gdt)
{
        unsigned int i;

        /* The default entries from setup_default_gdt_entries() are not
         * replaced.  See ignored_gdt() above. */
        for (i = 0; i < GDT_ENTRIES; i++)
              if (!ignored_gdt(i))
                    gdt[i] = cpu->arch.gdt[i];
}
```

switcher code

Understand this, and you understand the heart of our journey.

[drivers/lguest/x86/switcher_32.S] contains the low-level code which changes the CPU to run the Guest code, and returns to the Host when something happens.

We mark the start of the code to copy. It's placed in .text tho it's never run here. You'll see the trick macro at the end. Which interleaves data and text to effect.

```
.text
ENTRY(start_switcher_text)
```

When we reach switch_to_guest we have just left the safe and comforting shores of C code

%eax has the **"struct lguest_pages"** to use. Where we save state and still see it from the Guest.

And **%ebx** holds the **Guest shadow pagetable**: Once set we have truly left Host behind.

switch_to_guest

```
ENTRY(switch_to_guest)

    // We told gcc all its regs could fade,// Clobbered by our journey into the Guest
    // We could have saved them, if we tried // But time is our master and cycles count.
    // Segment registers must be saved for the Host
    // We push them on the Host stack for later
    pushl   %es
    pushl   %ds
    pushl   %gs
    pushl   %fs

    // But the compiler is fickle, and heeds . No warning of %ebp clobbers
    // When frame pointers are used.  That register
    // Must be saved and restored or chaos strikes.
    pushl   %ebp

    // The Host's stack is done, now save it away .  In our "struct lguest_pages" at
    // offset  distilled into asm-offsets.h
    movl    %esp, LGUEST_PAGES_host_sp(%eax)
```

switch_to_guest

// All saved and there's now **five steps before us:**
// Stack, GDT, IDT, TSS , Then last of all the page tables are flipped.
// Yet beware that our stack pointer must be // Always valid lest an NMI hits
// %edx does the duty here as we juggle // %eax is lguest_pages: our stack lies within.

```
        movl    %eax, %edx
        addl    $LGUEST_PAGES_regs, %edx
        movl    %edx, %esp
```

// The Guest's GDT we so carefully Placed in the "struct lguest_pages" before
```
        lgdt LGUEST_PAGES_guest_gdt_desc(%eax)
```

// The Guest's IDT we did partially copy to "struct lguest_pages" as well.
```
        lidt  LGUEST_PAGES_guest_idt_desc(%eax)
```

/ The TSS entry which controls traps must be loaded up with "ltr" now:
The GDT entry that TSS uses changes type when we load it: damn Intel!
For after we switch over our page tables That entry will be read-only: we'd crash.
```
        movl    $(GDT_ENTRY_TSS*8), %edx
        ltr   %dx
```

switch_to_guest

The Host's TSS entry was also marked used; Let's clear it again for our return.
The GDT descriptor of the Host points to the table after two "size" bytes

```
movl    (LGUEST_PAGES_host_gdt_desc+2)(%eax), %edx
```

Clear "used" from type field (byte 5, bit 2)

```
andb    $0xFD, (GDT_ENTRY_TSS*8 + 5)(%edx)
```

Once our page table's switched, the Guest is live! The Host fades as we run this final step.
Our "struct lguest_pages" is now read-only.

```
movl    %ebx, %cr3
```

The page table change did one tricky thing: **The Guest's register page has been mapped writable under our %esp** (stack) -- We can simply pop off all Guest regs.

```
popl %eax
popl %ebx
popl %ecx
popl %edx
popl %esi
popl %edi
popl %ebp
popl %gs
popl %fs
popl %ds
popl %es
```

switch_to_guest

// Near the base of the stack lurk two strange fields // Which we fill as we exit the Guest // These are the **trap number and its error**
// We can simply step past them on our way.
addl $8, %esp

// **The last five stack slots hold return address. And everything needed to switch privilege // From Switcher's level 0 to Guest's 1, // And the stack where the Guest had last left it. // Interrupts are turned back on: we are Guest.**
iret

//check **struct lguest_regs**

default_idt_entries

```
.data
.global default_idt_entries
default_idt_entries:
.text

        // The first two traps go straight back to the Host
        IRQ_STUBS 0 1 return_to_host

        IRQ_STUB 2 handle_nmi    // We'll say nothing, yet, about NMI

        IRQ_STUBS 3 31 return_to_host     // Other traps also return to the Host

        IRQ_STUBS 32 127 deliver_to_host      // All interrupts go via their handlers

        // 'Cept system calls coming from userspace
        // Are to go to the Guest, never the Host.
        IRQ_STUB 128 return_to_host

        IRQ_STUBS 129 255 deliver_to_host
```

default_idt_entries

```
.macro IRQ_STUB N TARGET
        .data; .long 1f; .text; 1:
 // Trap eight, ten through fourteen and seventeen supply an error number.  Else zero.
 .if (\N <> 8) && (\N < 10 || \N > 14) && (\N <> 17)
        pushl    $0
 .endif
        pushl    $\N
        jmp \TARGET
        ALIGN
.endm

// This macro creates numerous entries
.macro IRQ_STUBS FIRST LAST TARGET
 irq=\FIRST
 .rept \LAST-\FIRST+1
        IRQ_STUB irq \TARGET
  irq=irq+1
 .endr   .endm
```

return_to_host

The first path is trod when the Guest has trapped:

(Which trap it was has been pushed on the stack).

We need only switch back, and the Host will decode why we came home, and what needs to be done.

```
return_to_host:
        SWITCH_TO_HOST
        iret
```

deliver_to_host

We are lead to the second path like so: An interrupt, with some cause external. But now we must go home via that place where that interrupt was supposed to go. Here we see the trickness of run_guest_once(): The Host stack is formed like an interrupt, with EIP, CS and EFLAGS layered. Interrupt handlers end with "iret", And that will take us home at long long last.

deliver_to_host:
    SWITCH_TO_HOST

But first we must find the handler to call! The IDT descriptor for the Host has two bytes for size, and four for address: %edx will hold it for us for now.
    movl (LGUEST_PAGES_host_idt_desc+2)(%eax), %edx

We now know the table address we need, And saved the trap's number inside %ebx. Yet the pointer to the handler is smeared Across the bits of the table entry. What oracle can tell us how to extract from such a convoluted encoding? I consulted gcc, and it gave these instructions, which I gladly credit:

```
    leal  (%edx,%ebx,8), %eax
    movzwl  (%eax),%edx
    movl    4(%eax), %eax
    xorw    %ax, %ax
    orl   %eax, %edx
```

Now the address of the handler's in %edx . We call it now: its "iret" drops us home.

```
    jmp *%edx
```

SWITCH_TO_HOST

We tread two paths to switch back to the Host // Yet both must save Guest state and restore Host **So we put the routine in a macro(SWITCH_TO_HOST)**.

We save the Guest state: all registers first Laid out just as "struct lguest_regs" defines

```
#define SWITCH_TO_HOST                          \
        pushl   %es;                            \
        pushl   %ds;                            \
        pushl   %fs;                            \
        pushl   %gs;                            \
        pushl   %ebp;                           \
        pushl   %edi;                           \
        pushl   %esi;                           \
        pushl   %edx;                           \
        pushl   %ecx;                           \
        pushl   %ebx;                           \
        pushl   %eax;                           \
```

SWITCH_TO_HOST

Our stack and our code are using segments Set in the TSS and IDT
Load the lguest ds segment for now.

```
        movl    $(LGUEST_DS), %eax;             \
        movl    %eax, %ds;                      \
```

So where are we? Which CPU, which struct?
The stack is our clue: our TSS starts It at the end of "struct lguest_pages".
Or we may have stumbled while restoring our Guest segment regs while in switch_to_guest,
The fault pushed atop that part-unwound stack.
If we round the stack down to the page start we're at the start of "struct lguest_pages".

```
        movl    %esp, %eax;                     \
        andl $(~(1 << PAGE_SHIFT - 1)), %eax;   \
```

Save our trap number: the switch will obscure it (In the Host the Guest regs are not mapped here) %ebx holds it safe for deliver_to_host

```
        movl    LGUEST_PAGES_regs_trapnum(%eax), %ebx;
```

SWITCH_TO_HOST

The Host GDT, IDT and stack lie safely hidden from the Guest: We must return to the Host page tables (Hence that was saved in struct lguest_pages)

```
movl    LGUEST_PAGES_host_cr3(%eax), %edx;              \
movl    %edx, %cr3;                                     \
```

As before, when we looked back at the Host as we left and **marked TSS** unused, so must we now for the Guest left behind.

```
andb    $0xFD, (LGUEST_PAGES_guest_gdt+GDT_ENTRY_TSS*8+5)(%eax);    \

/* Switch to Host's GDT, IDT.*/
lgdt LGUEST_PAGES_host_gdt_desc(%eax);                  \
lidt  LGUEST_PAGES_host_idt_desc(%eax);                 \

/* Restore the Host's stack where its saved regs lie */     \
movl    LGUEST_PAGES_host_sp(%eax), %esp;               \
/* Last the TSS: our Host is returned */            \
movl    $(GDT_ENTRY_TSS*8), %edx;                       \
ltr    %dx;                                             \

/* Restore now the regs saved right at the first. */    \
popl %ebp;                                              \
popl %fs;                                               \
popl %gs;                                               \
popl %ds;                                               \
popl %es                                                \
```

THE END

drivers/lguest/lg.h

```
struct pgdir
{
	pgd_t *gpgdir;
	pgd_t *pgdir;
};

/* We have two pages shared with guests, per cpu.  */
struct lguest_pages
{
	/* This is the stack page mapped rw in guest */
	char spare[PAGE_SIZE - sizeof(struct lguest_regs)];
	struct lguest_regs regs;

	/* This is the host state & guest descriptor page, ro in guest */
	struct lguest_ro_state state;
} __attribute__((aligned(PAGE_SIZE)));
```

drivers/lguest/lg.h

```
struct lg_cpu {
        unsigned int id;        struct lguest *lg;      struct task_struct *tsk;
        struct mm_struct *mm;    /* == tsk->mm, but that becomes NULL on exit */

        u32 cr2;   int ts;       u32 esp1;       u8 ss1;

        /* Bitmap of what has changed: see CHANGED_* above. */
        int changed;

        unsigned long pending_notify; /* pfn from LHCALL_NOTIFY */

        /* At end of a page shared mapped over lguest_pages in guest.  */
        unsigned long regs_page;      struct lguest_regs *regs;

        struct lguest_pages *last_pages;
        int cpu_pgd;      /* which pgd this cpu is currently using */

        /* If a hypercall was asked for, this points to the arguments. */
        struct hcall_args *hcall;   u32 next_hcall;

        /* Virtual clock device */
        struct hrtimer hrt;

        /* Do we need to stop what we're doing and return to userspace? */
        int break_out;
        wait_queue_head_t break_wq;
        int halted;
```

drivers/lguest/lg.h

```
/* The private info the thread maintains about the guest. */
struct lguest
{
	struct lguest_data __user *lguest_data;
	struct lg_cpu cpus[NR_CPUS];
	unsigned int nr_cpus;

	u32 pfn_limit;
	/* This provides the offset to the base of guest-physical
	 * memory in the Launcher. */
	void __user *mem_base;
	unsigned long kernel_address;

	struct pgdir pgdirs[4];

	unsigned long noirq_start, noirq_end;

	unsigned int stack_pages;
	u32 tsc_khz;

	/* Dead? */
	const char *dead;
};
```

include/asm/lguest.h

```
struct lguest_regs {
        /* Manually saved part. */
        unsigned long eax, ebx, ecx, edx;
        unsigned long esi, edi, ebp;
        unsigned long gs;
        unsigned long fs, ds, es;
        unsigned long trapnum, errcode;
        /* Trap pushed part */
        unsigned long eip;
        unsigned long cs;
        unsigned long eflags;
        unsigned long esp;
        unsigned long ss;
};

struct lg_cpu_arch {
        /* The GDT entries copied into lguest_ro_state when running. */
        struct desc_struct gdt[GDT_ENTRIES];

        /* The IDT entries: some copied into lguest_ro_state when running. */
        struct desc_struct idt[IDT_ENTRIES];

        /* The address of the last guest-visible pagefault (ie. cr2). */
        unsigned long last_pagefault;
};
```

include/asm/lguest.h

```
/* This is a guest-specific page (mapped ro) into the guest. */
struct lguest_ro_state {
        /* Host information we need to restore when we switch back. */
        u32 host_cr3;
        struct desc_ptr host_idt_desc;
        struct desc_ptr host_gdt_desc;
        u32 host_sp;

        /* Fields which are used when guest is running. */
        struct desc_ptr guest_idt_desc;
        struct desc_ptr guest_gdt_desc;
        struct x86_hw_tss guest_tss;
        struct desc_struct guest_idt[IDT_ENTRIES];
        struct desc_struct guest_gdt[GDT_ENTRIES];
};
```

include/linux/lguest.h

```
/*G:032 The second method of communicating with the Host is to via "struct
 * lguest_data".  Once the Guest's initialization hypercall tells the Host where
 * this is, the Guest and Host both publish information in it. :*/
struct lguest_data
{
        /* 512 == enabled (same as eflags in normal hardware).  The Guest
         * changes interrupts so often that a hypercall is too slow. */
        unsigned int irq_enabled;
        /* Fine-grained interrupt disabling by the Guest */
        DECLARE_BITMAP(blocked_interrupts, LGUEST_IRQS);

        /* The Host writes the virtual address of the last page fault here,
         * which saves the Guest a hypercall.  CR2 is the native register where
         * this address would normally be found. */
        unsigned long cr2;

        /* Wallclock time set by the Host. */
        struct timespec time;

        /* Async hypercall ring.  Instead of directly making hypercalls, we can
         * place them in here for processing the next time the Host wants.
         * This batching can be quite efficient. */

        /* 0xFF == done (set by Host), 0 == pending (set by Guest). */
        u8 hcall_status[LHCALL_RING_SIZE];
        /* The actual registers for the hypercalls. */
        struct hcall_args hcalls[LHCALL_RING_SIZE];
```