

[RFC PATCH 34/35] Add the Xen virtual network device driver.

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2006-05/msg01928.html>

- *From:* Chris Wright <chrisw@xxxxxxxxxxxxx>
 - *Date:* Tue, 09 May 2006 00:00:34 -0700
-

The network device frontend driver allows the kernel to access network devices exported by a virtual machine containing a physical network device driver.

Signed-off-by: Ian Pratt <ian.pratt@xxxxxxxxxxxxx>
Signed-off-by: Christian Limpach <Christian.Limpach@xxxxxxxxxxxxx>
Signed-off-by: Chris Wright <chrisw@xxxxxxxxxxxxx>
Cc: netdev@xxxxxxxxxxxxx

TODO:

- drop proc
- more ethtool ops
- s/g support

drivers/net/Kconfig | 2
drivers/xen/Kconfig.net | 14
drivers/xen/Makefile | 3
drivers/xen/net_driver_util.c | 58 +
drivers/xen/netfront/Makefile | 4
drivers/xen/netfront/netfront.c | 1510 +++++
include/xen/net_driver_util.h | 48 +
7 files changed, 1639 insertions(+)

--- linux-2.6.orig/drivers/net/Kconfig
+++ linux-2.6/drivers/net/Kconfig
@@ -2325,6 +2325,8 @@ source "drivers/atm/Kconfig"

source "drivers/s390/net/Kconfig"

+source "drivers/xen/Kconfig.net"
+
config ISERIES_VETH
tristate "iSeries Virtual Ethernet driver support"
depends on PPC_ISERIES
--- linux-2.6.orig/drivers/xen/Makefile
+++ linux-2.6/drivers/xen/Makefile
@@ -1,7 +1,10 @@

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+obj-y += net_driver_util.o
obj-y += util.o

obj-y += core/
obj-y += console/
obj-y += xenbus/

+obj-$(CONFIG_XEN_NETDEV_FRONTEND) += netfront/
+
--- /dev/null
+++ linux-2.6/drivers/xen/Kconfig.net
@@ -0,0 +1,14 @@
+menu "Xen network device drivers"
+ depends on NETDEVICES && XEN
+
+config XEN_NETDEV_FRONTEND
+ tristate "Network-device frontend driver"
+ depends on XEN
+ default y
+ help
+ The network-device frontend driver allows the kernel to access
+ network interfaces within another guest OS. Unless you are building a
+ dedicated device-driver domain, or your master control domain
+ (domain 0), then you almost certainly want to say Y here.
+
+endmenu
--- /dev/null
+++ linux-2.6/drivers/xen/net_driver_util.c
@@ -0,0 +1,58 @@
+/******
+ *
+ * Utility functions for Xen network devices.
+ *
+ * Copyright (c) 2005 XenSource Ltd.
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public License version 2
+ * as published by the Free Software Foundation; or, when distributed
+ * separately from the Linux kernel or incorporated into other
+ * software packages, subject to the following license:
+ *
+ * Permission is hereby granted, free of charge, to any person obtaining a
+ * copy of this source file (the "Software"), to deal in the Software without
+ * restriction, including without limitation the rights to use, copy, modify,
+ * merge, publish, distribute, sublicense, and/or sell copies of the Software,
+ * and to permit persons to whom the Software is furnished to do so, subject
+ * to the following conditions:
+ *
+ * The above copyright notice and this permission notice shall be included in
+ * all copies or substantial portions of the Software.
+ *
+ */
```


[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ * Copyright (c) 2005, XenSource Ltd
+ *
+ * This program is free software; you can redistribute it and/or
+ * modify it under the terms of the GNU General Public License version 2
+ * as published by the Free Software Foundation; or, when distributed
+ * separately from the Linux kernel or incorporated into other
+ * software packages, subject to the following license:
+ *
+ * Permission is hereby granted, free of charge, to any person obtaining a copy
+ * of this source file (the "Software"), to deal in the Software without
+ * restriction, including without limitation the rights to use, copy, modify,
+ * merge, publish, distribute, sublicense, and/or sell copies of the Software,
+ * and to permit persons to whom the Software is furnished to do so, subject to
+ * the following conditions:
+ *
+ * The above copyright notice and this permission notice shall be included in
+ * all copies or substantial portions of the Software.
+ *
+ * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
+ * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
+ * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
+ * THE
+ * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
+ * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
+ * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
+ * DEALINGS
+ * IN THE SOFTWARE.
+ */
+
+#include <linux/config.h>
+#include <linux/module.h>
+#include <linux/version.h>
+#include <linux/kernel.h>
+#include <linux/sched.h>
+#include <linux/slab.h>
+#include <linux/string.h>
+#include <linux/errno.h>
+#include <linux/netdevice.h>
+#include <linux/inetdevice.h>
+#include <linux/etherdevice.h>
+#include <linux/skbuff.h>
+#include <linux/init.h>
+#include <linux/bitops.h>
+#include <linux/proc_fs.h>
+#include <linux/ethtool.h>
+#include <linux/in.h>
+#include <net/sock.h>
+#include <net/pkt_sched.h>
+#include <net/arp.h>
+#include <net/route.h>
+#include <asm/io.h>
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+#include <asm/uaccess.h>
+#include <xen/evtchn.h>
+#include <xen/xenbus.h>
+#include <xen/interface/io/netif.h>
+#include <xen/interface/memory.h>
+#ifdef CONFIG_XEN_BALLOON
+#include <xen/balloon.h>
+#endif
+#include <asm/page.h>
+#include <asm/uaccess.h>
+#include <xen/interface/grant_table.h>
+#include <xen/gnttab.h>
+#include <xen/net_driver_util.h>
+
+#define GRANT_INVALID_REF 0
+
+#define NET_TX_RING_SIZE __RING_SIZE((struct netif_tx_sring *)0, PAGE_SIZE)
+#define NET_RX_RING_SIZE __RING_SIZE((struct netif_rx_sring *)0, PAGE_SIZE)
+
+static inline void init_skb_shinfo(struct sk_buff *skb)
+{
+ atomic_set(&(skb_shinfo(skb)->dataref), 1);
+ skb_shinfo(skb)->nr_frags = 0;
+ skb_shinfo(skb)->frag_list = NULL;
+}
+
+struct netfront_info
+{
+ struct list_head list;
+ struct net_device *netdev;
+
+ struct net_device_stats stats;
+ unsigned int tx_full;
+
+ struct netif_tx_front_ring tx;
+ struct netif_rx_front_ring rx;
+
+ spinlock_t tx_lock;
+ spinlock_t rx_lock;
+
+ unsigned int handle;
+ unsigned int evtchn, irq;
+
+ /* What is the status of our connection to the remote backend? */
+#define BEST_CLOSED 0
+#define BEST_DISCONNECTED 1
+#define BEST_CONNECTED 2
+ unsigned int backend_state;
+
+ /* Is this interface open or closed (down or up)? */
+#define UST_CLOSED 0
```

```

+ #define UST_OPEN 1
+ unsigned int user_state;
+
+ /* Receive-ring batched refills. */
+ #define RX_MIN_TARGET 8
+ #define RX_DFL_MIN_TARGET 64
+ #define RX_MAX_TARGET NET_RX_RING_SIZE
+ int rx_min_target, rx_max_target, rx_target;
+ struct sk_buff_head rx_batch;
+
+ struct timer_list rx_refill_timer;
+
+ /*
+ * {tx,rx}_skbs store outstanding skbuffs. The first entry in each
+ * array is an index into a chain of free entries.
+ */
+ struct sk_buff *tx_skbs[NET_TX_RING_SIZE+1];
+ struct sk_buff *rx_skbs[NET_RX_RING_SIZE+1];
+
+ grant_ref_t gref_tx_head;
+ grant_ref_t grant_tx_ref[NET_TX_RING_SIZE + 1];
+ grant_ref_t gref_rx_head;
+ grant_ref_t grant_rx_ref[NET_TX_RING_SIZE + 1];
+
+ struct xenbus_device *xbdev;
+ int tx_ring_ref;
+ int rx_ring_ref;
+ u8 mac[ETH_ALEN];
+
+ unsigned long rx_pfn_array[NET_RX_RING_SIZE];
+ struct multicall_entry rx_mcl[NET_RX_RING_SIZE+1];
+ struct mmu_update rx_mmu[NET_RX_RING_SIZE];
+ };
+
+ /*
+ * Access macros for acquiring freeing slots in {tx,rx}_skbs[].
+ */
+
+ static inline void add_id_to_freelist(struct sk_buff **list, unsigned short id)
+ {
+ list[id] = list[0];
+ list[0] = (void *) (unsigned long) id;
+ }
+
+ static inline unsigned short get_id_from_freelist(struct sk_buff **list)
+ {
+ unsigned int id = (unsigned int) (unsigned long) list[0];
+ list[0] = list[id];
+ return id;
+ }
+

```

```

+ #ifdef DEBUG
+ static char *be_state_name[] = {
+ [BEST_CLOSED] = "closed",
+ [BEST_DISCONNECTED] = "disconnected",
+ [BEST_CONNECTED] = "connected",
+ };
+ #endif
+
+ #define DPRINTK(fmt, args...) pr_debug("netfront (%s:%d) " fmt, \
+ __FUNCTION__, __LINE__, ##args)
+ #define IPRINTK(fmt, args...) \
+ printk(KERN_INFO "netfront: " fmt, ##args)
+ #define WPRINTK(fmt, args...) \
+ printk(KERN_WARNING "netfront: " fmt, ##args)
+
+
+ static int talk_to_backend(struct xenbus_device *, struct netfront_info *);
+ static int setup_device(struct xenbus_device *, struct netfront_info *);
+ static int create_netdev(int, struct xenbus_device *, struct net_device **);
+
+ static void netfront_closing(struct xenbus_device *);
+
+ static void end_access(int, void *);
+ static void netif_disconnect_backend(struct netfront_info *);
+ static void close_netdev(struct netfront_info *);
+ static void netif_free(struct netfront_info *);
+
+ static void show_device(struct netfront_info *);
+
+ static void network_connect(struct net_device *);
+ static void network_tx_buf_gc(struct net_device *);
+ static void network_alloc_rx_buffers(struct net_device *);
+ static int send_fake_arp(struct net_device *);
+
+ static irqreturn_t netif_int(int irq, void *dev_id, struct pt_regs *ptregs);
+
+ #define XEN_XENNET_PROC_INTERFACE
+ #if defined(CONFIG_PROC_FS) && defined(XEN_XENNET_PROC_INTERFACE)
+ static int xennet_proc_init(void);
+ static int xennet_proc_addif(struct net_device *dev);
+ static void xennet_proc_delif(struct net_device *dev);
+ #else
+ #define xennet_proc_init() (0)
+ #define xennet_proc_addif(d) (0)
+ #define xennet_proc_delif(d) ((void)0)
+ #endif
+
+
+ /**
+ * Entry point to this code when a new device is created. Allocate the basic
+ * structures and the ring buffers for communication with the backend, and

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ * inform the backend of the appropriate details for those. Switch to
+ * Connected state.
+ */
+static int netfront_probe(struct xenbus_device *dev,
+ const struct xenbus_device_id *id)
+{
+ int err;
+ struct net_device *netdev;
+ struct netfront_info *info;
+ unsigned int handle;
+
+ err = xenbus_scanf(XBT_NULL, dev->nodename, "handle", "%u", &handle);
+ if (err != 1) {
+ xenbus_dev_fatal(dev, err, "reading handle");
+ return err;
+ }
+
+ err = create_netdev(handle, dev, &netdev);
+ if (err) {
+ xenbus_dev_fatal(dev, err, "creating netdev");
+ return err;
+ }
+
+ info = netdev_priv(netdev);
+ dev->data = info;
+
+ err = talk_to_backend(dev, info);
+ if (err) {
+ kfree(info);
+ dev->data = NULL;
+ return err;
+ }
+
+ return 0;
+}
+
+/**
+ * We are reconnecting to the backend, due to a suspend/resume, or a backend
+ * driver restart. We tear down our netif structure and recreate it, but
+ * leave the device-layer structures intact so that this is transparent to the
+ * rest of the kernel.
+ */
+static int netfront_resume(struct xenbus_device *dev)
+{
+ struct netfront_info *info = dev->data;
+
+ DPRINTK("%s\n", dev->nodename);
+
+ netif_disconnect_backend(info);
+ return talk_to_backend(dev, info);
+}
```

```

+}
+
+
+/* Common code used when first setting up, and when resuming. */
+static int talk_to_backend(struct xenbus_device *dev,
+ struct netfront_info *info)
+{
+ const char *message;
+ xenbus_transaction_t xbt;
+ int err;
+
+
+ err = xen_net_read_mac(dev, info->mac);
+ if (err) {
+ xenbus_dev_fatal(dev, err, "parsing %s/mac", dev->nodename);
+ goto out;
+ }
+
+
+ /* Create shared ring, alloc event channel. */
+ err = setup_device(dev, info);
+ if (err)
+ goto out;
+
+
+again:
+ err = xenbus_transaction_start(&xbt);
+ if (err) {
+ xenbus_dev_fatal(dev, err, "starting transaction");
+ goto destroy_ring;
+ }
+
+
+ err = xenbus_printf(xbt, dev->nodename, "tx-ring-ref", "%u",
+ info->tx_ring_ref);
+ if (err) {
+ message = "writing tx ring-ref";
+ goto abort_transaction;
+ }
+
+ err = xenbus_printf(xbt, dev->nodename, "rx-ring-ref", "%u",
+ info->rx_ring_ref);
+ if (err) {
+ message = "writing rx ring-ref";
+ goto abort_transaction;
+ }
+
+ err = xenbus_printf(xbt, dev->nodename,
+ "event-channel", "%u", info->evtchn);
+ if (err) {
+ message = "writing event-channel";
+ goto abort_transaction;
+ }
+
+
+ err = xenbus_printf(xbt, dev->nodename,
+ "state", "%d", XenbusStateConnected);
+ if (err) {

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ message = "writing frontend XenbusStateConnected";
+ goto abort_transaction;
+ }
+
+ err = xenbus_transaction_end(xbt, 0);
+ if (err) {
+ if (err == -EAGAIN)
+ goto again;
+ xenbus_dev_fatal(dev, err, "completing transaction");
+ goto destroy_ring;
+ }
+
+ return 0;
+
+ abort_transaction:
+ xenbus_transaction_end(xbt, 1);
+ xenbus_dev_fatal(dev, err, "%s", message);
+ destroy_ring:
+ netif_free(info);
+ out:
+ return err;
+}
+
+
+static int setup_device(struct xenbus_device *dev, struct netfront_info *info)
+{
+ struct netif_tx_sring *txs;
+ struct netif_rx_sring *rxs;
+ int err;
+ struct net_device *netdev = info->netdev;
+
+ info->tx_ring_ref = GRANT_INVALID_REF;
+ info->rx_ring_ref = GRANT_INVALID_REF;
+ info->rx.sring = NULL;
+ info->tx.sring = NULL;
+ info->irq = 0;
+
+ txs = (struct netif_tx_sring *)get_zeroed_page(GFP_KERNEL);
+ if (!txs) {
+ err = -ENOMEM;
+ xenbus_dev_fatal(dev, err, "allocating tx ring page");
+ goto fail;
+ }
+ rxs = (struct netif_rx_sring *)get_zeroed_page(GFP_KERNEL);
+ if (!rxs) {
+ err = -ENOMEM;
+ xenbus_dev_fatal(dev, err, "allocating rx ring page");
+ free_page((unsigned long)txs);
+ goto fail;
+ }
+ info->backend_state = BEST_DISCONNECTED;
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+
+ SHARED_RING_INIT(txs);
+ FRONT_RING_INIT(&info->tx, txs, PAGE_SIZE);
+
+ SHARED_RING_INIT(rxs);
+ FRONT_RING_INIT(&info->rx, rxs, PAGE_SIZE);
+
+ err = xenbus_grant_ring(dev, virt_to_mfn(txs));
+ if (err < 0)
+ goto fail;
+ info->tx_ring_ref = err;
+
+ err = xenbus_grant_ring(dev, virt_to_mfn(rxs));
+ if (err < 0)
+ goto fail;
+ info->rx_ring_ref = err;
+
+ err = xenbus_alloc_evtchn(dev, &info->evtchn);
+ if (err)
+ goto fail;
+
+ memcpy(netdev->dev_addr, info->mac, ETH_ALEN);
+ network_connect(netdev);
+ info->irq = bind_evtchn_to_irqhandler(
+ info->evtchn, netif_int, SA_SAMPLE_RANDOM, netdev->name,
+ netdev);
+ (void)send_fake_arp(netdev);
+ show_device(info);
+
+ return 0;
+
+ fail:
+ netif_free(info);
+ return err;
+}
+
+/**
+ * Callback received when the backend's state changes.
+ */
+static void backend_changed(struct xenbus_device *dev,
+ XenbusState backend_state)
+{
+ DPRINTK("\n");
+
+ switch (backend_state) {
+ case XenbusStateInitialising:
+ case XenbusStateInitWait:
+ case XenbusStateInitialised:
+ case XenbusStateConnected:
+ case XenbusStateUnknown:
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ case XenbusStateClosed:
+ break;
+
+ case XenbusStateClosing:
+ netfront_closing(dev);
+ break;
+ }
+}
+
+
+/** Send a packet on a net device to encourage switches to learn the
+ * MAC. We send a fake ARP request.
+ *
+ * @param dev device
+ * @return 0 on success, error code otherwise
+ */
+static int send_fake_arp(struct net_device *dev)
+{
+ struct sk_buff *skb;
+ u32 src_ip, dst_ip;
+
+ dst_ip = INADDR_BROADCAST;
+ src_ip = inet_select_addr(dev, dst_ip, RT_SCOPE_LINK);
+
+ /* No IP? Then nothing to do. */
+ if (src_ip == 0)
+ return 0;
+
+ skb = arp_create(ARPOP_REPLY, ETH_P_ARP,
+ dst_ip, dev, src_ip,
+ /*dst_hw*/ NULL, /*src_hw*/ NULL,
+ /*target_hw*/ dev->dev_addr);
+ if (skb == NULL)
+ return -ENOMEM;
+
+ return dev_queue_xmit(skb);
+}
+
+
+static int network_open(struct net_device *dev)
+{
+ struct netfront_info *np = netdev_priv(dev);
+
+ memset(&np->stats, 0, sizeof(np->stats));
+
+ np->user_state = UST_OPEN;
+
+ network_alloc_rx_buffers(dev);
+ np->rx.sring->rsp_event = np->rx.rsp_cons + 1;
+
+ netif_start_queue(dev);
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+
+ return 0;
+}
+
+static void network_tx_buf_gc(struct net_device *dev)
+{
+ RING_IDX i, prod;
+ unsigned short id;
+ struct netfront_info *np = netdev_priv(dev);
+ struct sk_buff *skb;
+
+ if (np->backend_state != BEST_CONNECTED)
+ return;
+
+ do {
+ prod = np->tx.sring->rsp_prod;
+ rmb(); /* Ensure we see responses up to 'rp'. */
+
+ for (i = np->tx.rsp_cons; i != prod; i++) {
+ id = RING_GET_RESPONSE(&np->tx, i)->id;
+ skb = np->tx.skbs[id];
+ if (unlikely(gnttab_query_foreign_access(
+ np->grant_tx_ref[id]) != 0)) {
+ printk(KERN_ALERT "network_tx_buf_gc: warning "
+ "-- grant still in use by backend "
+ "domain.\n");
+ goto out;
+ }
+ gnttab_end_foreign_access_ref(
+ np->grant_tx_ref[id], GNTMAP_readonly);
+ gnttab_release_grant_reference(
+ &np->gref_tx_head, np->grant_tx_ref[id]);
+ np->grant_tx_ref[id] = GRANT_INVALID_REF;
+ add_id_to_freelist(np->tx.skbs, id);
+ dev_kfree_skb_irq(skb);
+ }
+
+ np->tx.rsp_cons = prod;
+
+ /*
+ * Set a new event, then check for race with update of tx_cons.
+ * Note that it is essential to schedule a callback, no matter
+ * how few buffers are pending. Even if there is space in the
+ * transmit ring, higher layers may be blocked because too much
+ * data is outstanding: in such cases notification from Xen is
+ * likely to be the only kick that we'll get.
+ */
+ np->tx.sring->rsp_event =
+ prod + ((np->tx.sring->req_prod - prod) >> 1) + 1;
+ mb();
+ } while (prod != np->tx.sring->rsp_prod);
```

```

+
+ out:
+ if (np->tx_full &&
+ ((np->tx.sring->req_prod - prod) < NET_TX_RING_SIZE)) {
+ np->tx_full = 0;
+ if (np->user_state == UST_OPEN)
+ netif_wake_queue(dev);
+ }
+ }
+
+
+static void rx_refill_timeout(unsigned long data)
+{
+ struct net_device *dev = (struct net_device *)data;
+ netif_rx_schedule(dev);
+ }
+
+
+static void network_alloc_rx_buffers(struct net_device *dev)
+{
+ unsigned short id;
+ struct netfront_info *np = netdev_priv(dev);
+ struct sk_buff *skb;
+ int i, batch_target;
+ RING_IDX req_prod = np->rx.req_prod_pvt;
+ struct xen_memory_reservation reservation;
+ grant_ref_t ref;
+
+ if (unlikely(np->backend_state != BEST_CONNECTED))
+ return;
+
+ /*
+ * Allocate skbuffs greedily, even though we batch updates to the
+ * receive ring. This creates a less bursty demand on the memory
+ * allocator, so should reduce the chance of failed allocation requests
+ * both for ourself and for other kernel subsystems.
+ */
+ batch_target = np->rx_target - (req_prod - np->rx.rsp_cons);
+ for (i = skb_queue_len(&np->rx_batch); i < batch_target; i++) {
+ /*
+ * Subtract dev_alloc_skb headroom (16 bytes) and shared info
+ * tailroom then round down to SKB_DATA_ALIGN boundary.
+ */
+ skb = __dev_alloc_skb(
+ ((PAGE_SIZE - sizeof(struct skb_shared_info)) &
+ (-SKB_DATA_ALIGN(1))) - 16,
+ GFP_ATOMIC|__GFP_NOWARN);
+ if (skb == NULL) {
+ /* Any skbuffs queued for refill? Force them out. */
+ if (i != 0)
+ goto refill;

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```

+ /* Could not allocate any skbuffs. Try again later. */
+ mod_timer(&np->rx_refill_timer,
+ jiffies + (HZ/10));
+ return;
+ }
+ __skb_queue_tail(&np->rx_batch, skb);
+ }
+
+ /* Is the batch large enough to be worthwhile? */
+ if (i < (np->rx_target/2))
+ return;
+
+ /* Adjust our fill target if we risked running out of buffers. */
+ if (((req_prod - np->rx.sring->rsp_prod) < (np->rx_target / 4)) &&
+ ((np->rx_target * 2) > np->rx_max_target))
+ np->rx_target = np->rx_max_target;
+
+ refill:
+ for (i = 0; ; i++) {
+ if ((skb = __skb_dequeue(&np->rx_batch)) == NULL)
+ break;
+
+ skb->dev = dev;
+
+ id = get_id_from_freelist(np->rx_skbs);
+
+ np->rx_skbs[id] = skb;
+
+ RING_GET_REQUEST(&np->rx, req_prod + i)->id = id;
+ ref = gnttab_claim_grant_reference(&np->gref_rx_head);
+ BUG_ON((signed short)ref < 0);
+ np->grant_rx_ref[id] = ref;
+ gnttab_grant_foreign_transfer_ref(ref,
+ np->xbdev->otherend_id,
+ __pa(skb->head) >> PAGE_SHIFT);
+ RING_GET_REQUEST(&np->rx, req_prod + i)->gref = ref;
+ np->rx_pfn_array[i] = virt_to_mfn(skb->head);
+
+ #ifndef CONFIG_XEN_SHADOW_MODE
+ if (!xen_feature(XENFEAT_auto_translated_physmap)) {
+ /* Remove this page before passing back to Xen. */
+ set_phys_to_machine(__pa(skb->head) >> PAGE_SHIFT,
+ INVALID_P2M_ENTRY);
+ MULTI_update_va_mapping(np->rx_mcl+i,
+ (unsigned long)skb->head,
+ __pte(0), 0);
+ }
+ #endif
+ }
+
+ #ifdef CONFIG_XEN_BALLOON

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ /* Tell the balloon driver what is going on. */
+ balloon_update_driver_allowance(i);
+ #endif
+
+ reservation.extent_start = np->rx_pfn_array;
+ reservation.nr_extents = i;
+ reservation.extent_order = 0;
+ reservation.address_bits = 0;
+ reservation.domid = DOMID_SELF;
+
+ #ifndef CONFIG_XEN_SHADOW_MODE
+ if (!xen_feature(XENFEAT_auto_translated_physmap)) {
+ /* After all PTEs have been zapped, flush the TLB. */
+ np->rx_mcl[i-1].args[MULTI_UVMFLAGS_INDEX] =
+ UVMF_TLB_FLUSH|UVMF_ALL;
+
+ /* Give away a batch of pages. */
+ np->rx_mcl[i].op = __HYPERVISOR_memory_op;
+ np->rx_mcl[i].args[0] = XENMEM_decrease_reservation;
+ np->rx_mcl[i].args[1] = (unsigned long)&reservation;
+
+ /* Zap PTEs and give away pages in one big multicall. */
+ (void)HYPERVISOR_multicall(np->rx_mcl, i+1);
+
+ /* Check return status of HYPERVISOR_memory_op(). */
+ if (unlikely(np->rx_mcl[i].result != i))
+ panic("Unable to reduce memory reservation\n");
+ } else
+ #endif
+ if (HYPERVISOR_memory_op(XENMEM_decrease_reservation,
+ &reservation) != i)
+ panic("Unable to reduce memory reservation\n");
+
+ /* Above is a suitable barrier to ensure backend will see requests. */
+ np->rx.req_prod_pvt = req_prod + i;
+ RING_PUSH_REQUESTS(&np->rx);
+ }
+
+
+ static int network_start_xmit(struct sk_buff *skb, struct net_device *dev)
+ {
+ unsigned short id;
+ struct netfront_info *np = netdev_priv(dev);
+ struct netif_tx_request *tx;
+ RING_IDX i;
+ grant_ref_t ref;
+ unsigned long mfn;
+ int notify;
+
+ if (unlikely(np->tx_full)) {
+ printk(KERN_ALERT "%s: full queue wasn't stopped!\n",
```

```

+ dev->name);
+ netif_stop_queue(dev);
+ goto drop;
+ }
+
+ if (unlikely((((unsigned long)skb->data & ~PAGE_MASK) + skb->len) >=
+ PAGE_SIZE)) {
+ struct sk_buff *nskb;
+ nskb = __dev_alloc_skb(skb->len, GFP_ATOMIC|__GFP_NOWARN);
+ if (unlikely(nskb == NULL))
+ goto drop;
+ skb_put(nskb, skb->len);
+ memcpy(nskb->data, skb->data, skb->len);
+ nskb->dev = skb->dev;
+ dev_kfree_skb(skb);
+ skb = nskb;
+ }
+
+ spin_lock_irq(&np->tx_lock);
+
+ if (np->backend_state != BEST_CONNECTED) {
+ spin_unlock_irq(&np->tx_lock);
+ goto drop;
+ }
+
+ i = np->tx.req_prod_pvt;
+
+ id = get_id_from_freelist(np->tx_skbs);
+ np->tx_skbs[id] = skb;
+
+ tx = RING_GET_REQUEST(&np->tx, i);
+
+ tx->id = id;
+ ref = gnttab_claim_grant_reference(&np->gref_tx_head);
+ BUG_ON((signed short)ref < 0);
+ mfn = virt_to_mfn(skb->data);
+ gnttab_grant_foreign_access_ref(
+ ref, np->xbdev->otherend_id, mfn, GNTMAP_readonly);
+ tx->gref = np->grant_tx_ref[id] = ref;
+ tx->offset = (unsigned long)skb->data & ~PAGE_MASK;
+ tx->size = skb->len;
+ tx->flags = (skb->ip_summed == CHECKSUM_HW) ? NETTXF_csum_blank : 0;
+
+ np->tx.req_prod_pvt = i + 1;
+ RING_PUSH_REQUESTS_AND_CHECK_NOTIFY(&np->tx, notify);
+ if (notify)
+ notify_remote_via_irq(np->irq);
+
+ network_tx_buf_gc(dev);
+
+ if (RING_FULL(&np->tx)) {

```

```

+ np->tx_full = 1;
+ netif_stop_queue(dev);
+ }
+
+ spin_unlock_irq(&np->tx_lock);
+
+ np->stats.tx_bytes += skb->len;
+ np->stats.tx_packets++;
+
+ return 0;
+
+ drop:
+ np->stats.tx_dropped++;
+ dev_kfree_skb(skb);
+ return 0;
+}
+
+static irqreturn_t netif_int(int irq, void *dev_id, struct pt_regs *ptregs)
+{
+ struct net_device *dev = dev_id;
+ struct netfront_info *np = netdev_priv(dev);
+ unsigned long flags;
+
+ spin_lock_irqsave(&np->tx_lock, flags);
+ network_tx_buf_gc(dev);
+ spin_unlock_irqrestore(&np->tx_lock, flags);
+
+ if (RING_HAS_UNCONSUMED_RESPONSES(&np->rx) &&
+ (np->user_state == UST_OPEN))
+ netif_rx_schedule(dev);
+
+ return IRQ_HANDLED;
+}
+
+static int netif_poll(struct net_device *dev, int *pbudget)
+{
+ struct netfront_info *np = netdev_priv(dev);
+ struct sk_buff *skb, *nskb;
+ struct netif_rx_response *rx;
+ RING_IDX i, rp;
+ struct mmu_update *mmu = np->rx_mmu;
+ struct multicall_entry *mcl = np->rx_mcl;
+ int work_done, budget, more_to_do = 1;
+ struct sk_buff_head rxq;
+ unsigned long flags;
+ unsigned long mfn;
+ grant_ref_t ref;
+
+ spin_lock(&np->rx_lock);
+

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ if (np->backend_state != BEST_CONNECTED) {
+ spin_unlock(&np->rx_lock);
+ return 0;
+ }
+
+ skb_queue_head_init(&rxq);
+
+ if ((budget = *pbudget) > dev->quota)
+ budget = dev->quota;
+ rp = np->rx.sring->rsp_prod;
+ rmb(); /* Ensure we see queued responses up to 'rp'. */
+
+ for (i = np->rx.rsp_cons, work_done = 0;
+ (i != rp) && (work_done < budget);
+ i++, work_done++) {
+ rx = RING_GET_RESPONSE(&np->rx, i);
+
+ /*
+ * This definitely indicates a bug, either in this driver or
+ * in the backend driver. In future this should flag the bad
+ * situation to the system controller to reboot the backed.
+ */
+ if ((ref = np->grant_rx_ref[rx->id]) == GRANT_INVALID_REF) {
+ WPRINTK("Bad rx response id %d.\n", rx->id);
+ work_done--;
+ continue;
+ }
+
+ /* Memory pressure, insufficient buffer headroom, ... */
+ if ((mfn = gnttab_end_foreign_transfer_ref(ref)) == 0) {
+ if (net_ratelimit())
+ WPRINTK("Unfulfilled rx req (id=%d, st=%d).\n",
+ rx->id, rx->status);
+ RING_GET_REQUEST(&np->rx, np->rx.req_prod_pvt)->id =
+ rx->id;
+ RING_GET_REQUEST(&np->rx, np->rx.req_prod_pvt)->gref =
+ ref;
+ np->rx.req_prod_pvt++;
+ RING_PUSH_REQUESTS(&np->rx);
+ work_done--;
+ continue;
+ }
+
+ gnttab_release_grant_reference(&np->gref_rx_head, ref);
+ np->grant_rx_ref[rx->id] = GRANT_INVALID_REF;
+
+ skb = np->rx_skbs[rx->id];
+ add_id_to_freelist(np->rx_skbs, rx->id);
+
+ /* NB. We handle skb overflow later. */
+ skb->data = skb->head + rx->offset;
```

```

+ skb->len = rx->status;
+ skb->tail = skb->data + skb->len;
+
+ if (rx->flags & NETRXF_data_validated)
+ skb->ip_summed = CHECKSUM_UNNECESSARY;
+
+ np->stats.rx_packets++;
+ np->stats.rx_bytes += rx->status;
+
+ #ifndef CONFIG_XEN_SHADOW_MODE
+ if (!xen_feature(XENFEAT_auto_translated_physmap)) {
+ /* Remap the page. */
+ MULTI_update_va_mapping(mcl, (unsigned long)skb->head,
+ pfn_pte_ma(mfn, PAGE_KERNEL),
+ 0);
+ mcl++;
+ mmu->ptr = ((maddr_t)mfn << PAGE_SHIFT)
+ | MMU_MACHPHYS_UPDATE;
+ mmu->val = __pa(skb->head) >> PAGE_SHIFT;
+ mmu++;
+
+ set_phys_to_machine(__pa(skb->head) >> PAGE_SHIFT,
+ mfn);
+ }
+ #endif
+
+ __skb_queue_tail(&rxq, skb);
+ }
+
+ #ifdef CONFIG_XEN_BALLOON
+ /* Some pages are no longer absent... */
+ balloon_update_driver_allowance(-work_done);
+ #endif
+
+ /* Do all the remapping work, and M2P updates, in one big hypercall. */
+ if (likely((mcl - np->rx_mcl) != 0)) {
+ mcl->op = __HYPERVISOR_mmu_update;
+ mcl->args[0] = (unsigned long)np->rx_mmu;
+ mcl->args[1] = mmu - np->rx_mmu;
+ mcl->args[2] = 0;
+ mcl->args[3] = DOMID_SELF;
+ mcl++;
+ (void)HYPERVISOR_multicall(np->rx_mcl, mcl - np->rx_mcl);
+ }
+
+ while ((skb = __skb_dequeue(&rxq)) != NULL) {
+ if (skb->len > (dev->mtu + ETH_HLEN + 4)) {
+ if (net_ratelimit())
+ printk(KERN_INFO "Received packet too big for "
+ "MTU (%d > %d)\n",
+ skb->len - ETH_HLEN - 4, dev->mtu);

```

```

+ skb->len = 0;
+ skb->tail = skb->data;
+ init_skb_shinfo(skb);
+ dev_kfree_skb(skb);
+ continue;
+ }
+
+ /*
+ * Enough room in skbuff for the data we were passed? Also,
+ * Linux expects at least 16 bytes headroom in each rx buffer.
+ */
+ if (unlikely(skb->tail > skb->end) ||
+     unlikely((skb->data - skb->head) < 16)) {
+     if (net_ratelimit()) {
+         if (skb->tail > skb->end)
+             printk(KERN_INFO "Received packet "
+                    "is %zd bytes beyond tail.\n",
+                    skb->tail - skb->end);
+         else
+             printk(KERN_INFO "Received packet "
+                    "is %zd bytes before head.\n",
+                    16 - (skb->data - skb->head));
+     }
+ }
+
+ nskb = __dev_alloc_skb(skb->len + 2,
+                        GFP_ATOMIC|__GFP_NOWARN);
+ if (nskb != NULL) {
+     skb_reserve(nskb, 2);
+     skb_put(nskb, skb->len);
+     memcpy(nskb->data, skb->data, skb->len);
+     nskb->dev = skb->dev;
+     nskb->ip_summed = skb->ip_summed;
+ }
+
+ /* Reinitialise and then destroy the old skbuff. */
+ skb->len = 0;
+ skb->tail = skb->data;
+ init_skb_shinfo(skb);
+ dev_kfree_skb(skb);
+
+ /* Switch old for new, if we copied the buffer. */
+ if ((skb = nskb) == NULL)
+     continue;
+ }
+
+ /* Set the shinfo area, which is hidden behind the data. */
+ init_skb_shinfo(skb);
+ /* Ethernet work: Delayed to here as it peeks the header. */
+ skb->protocol = eth_type_trans(skb, dev);
+
+ /* Pass it up. */

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ netif_receive_skb(skb);
+ dev->last_rx = jiffies;
+ }
+
+ np->rx.rsp_cons = i;
+
+ /* If we get a callback with very few responses, reduce fill target. */
+ /* NB. Note exponential increase, linear decrease. */
+ if (((np->rx.req_prod_pvt - np->rx.sring->rsp_prod) >
+ ((3*np->rx_target) / 4)) &&
+ (--np->rx_target < np->rx_min_target))
+ np->rx_target = np->rx_min_target;
+
+ network_alloc_rx_buffers(dev);
+
+ *pbudget -= work_done;
+ dev->quota -= work_done;
+
+ if (work_done < budget) {
+ local_irq_save(flags);
+
+ RING_FINAL_CHECK_FOR_RESPONSES(&np->rx, more_to_do);
+ if (!more_to_do)
+ __netif_rx_complete(dev);
+
+ local_irq_restore(flags);
+ }
+
+ spin_unlock(&np->rx_lock);
+
+ return more_to_do;
+}
+
+
+static int network_close(struct net_device *dev)
+{
+ struct netfront_info *np = netdev_priv(dev);
+ np->user_state = UST_CLOSED;
+ netif_stop_queue(np->netdev);
+ return 0;
+}
+
+
+static struct net_device_stats *network_get_stats(struct net_device *dev)
+{
+ struct netfront_info *np = netdev_priv(dev);
+ return &np->stats;
+}
+
+
+static void network_connect(struct net_device *dev)
+{
```

```

+ struct netfront_info *np;
+ int i, requeue_idx;
+ struct netif_tx_request *tx;
+ struct sk_buff *skb;
+
+ np = netdev_priv(dev);
+ spin_lock_irq(&np->tx_lock);
+ spin_lock(&np->rx_lock);
+
+ /* Recovery procedure: */
+
+ /* Step 1: Reinitialise variables. */
+ np->tx_full = 0;
+
+ /*
+ * Step 2: Rebuild the RX and TX ring contents.
+ * NB. We could just free the queued TX packets now but we hope
+ * that sending them out might do some good. We have to rebuild
+ * the RX ring because some of our pages are currently flipped out
+ * so we can't just free the RX skbs.
+ * NB2. Freelist index entries are always going to be less than
+ * __PAGE_OFFSET, whereas pointers to skbs will always be equal or
+ * greater than __PAGE_OFFSET: we use this property to distinguish
+ * them.
+ */
+
+ /*
+ * Rebuild the TX buffer freelist and the TX ring itself.
+ * NB. This reorders packets. We could keep more private state
+ * to avoid this but maybe it doesn't matter so much given the
+ * interface has been down.
+ */
+ for (requeue_idx = 0, i = 1; i <= NET_TX_RING_SIZE; i++) {
+ if ((unsigned long)np->tx_skbs[i] < __PAGE_OFFSET)
+ continue;
+
+ skb = np->tx_skbs[i];
+
+ tx = RING_GET_REQUEST(&np->tx, requeue_idx);
+ requeue_idx++;
+
+ tx->id = i;
+ gnttab_grant_foreign_access_ref(
+ np->grant_tx_ref[i], np->xbdev->otherend_id,
+ virt_to_mfn(np->tx_skbs[i]->data),
+ GNTMAP_readonly);
+ tx->gref = np->grant_tx_ref[i];
+ tx->offset = (unsigned long)skb->data & ~PAGE_MASK;
+ tx->size = skb->len;
+ tx->flags = (skb->ip_summed == CHECKSUM_HW) ?
+ NETTXF_csum_blank : 0;

```

```

+
+ np->stats.tx_bytes += skb->len;
+ np->stats.tx_packets++;
+ }
+
+ np->tx.req_prod_pvt = requeue_idx;
+ RING_PUSH_REQUESTS(&np->tx);
+
+ /* Rebuild the RX buffer freelist and the RX ring itself. */
+ for (requeue_idx = 0, i = 1; i <= NET_RX_RING_SIZE; i++) {
+ if ((unsigned long)np->rx_skbs[i] < __PAGE_OFFSET)
+ continue;
+ gnttab_grant_foreign_transfer_ref(
+ np->grant_rx_ref[i], np->xbdev->otherend_id,
+ __pa(np->rx_skbs[i]->data) >> PAGE_SHIFT);
+ RING_GET_REQUEST(&np->rx, requeue_idx)->gref =
+ np->grant_rx_ref[i];
+ RING_GET_REQUEST(&np->rx, requeue_idx)->id = i;
+ requeue_idx++;
+ }
+
+ np->rx.req_prod_pvt = requeue_idx;
+ RING_PUSH_REQUESTS(&np->rx);
+
+ /*
+ * Step 3: All public and private state should now be sane. Get
+ * ready to start sending and receiving packets and give the driver
+ * domain a kick because we've probably just queued some
+ * packets.
+ */
+ np->backend_state = BEST_CONNECTED;
+ notify_remote_via_irq(np->irq);
+ network_tx_buf_gc(dev);
+
+ if (np->user_state == UST_OPEN)
+ netif_start_queue(dev);
+
+ spin_unlock(&np->rx_lock);
+ spin_unlock_irq(&np->tx_lock);
+ }
+
+static void show_device(struct netfront_info *np)
+{
+ #ifdef DEBUG
+ if (np) {
+ IPRINTK("<vif handle=%u %s(%s) evtchn=%u tx=%p rx=%p>\n",
+ np->handle,
+ be_state_name[np->backend_state],
+ np->user_state ? "open" : "closed",
+ np->evtchn,
+ np->tx,

```

```

+ np->rx);
+ } else
+ IPRINTK("<vif NULL>\n");
+ #endif
+ }
+
+ static void netif_uninit(struct net_device *dev)
+ {
+ struct netfront_info *np = netdev_priv(dev);
+ gnttab_free_grant_references(np->gref_tx_head);
+ gnttab_free_grant_references(np->gref_rx_head);
+ }
+
+ static struct ethtool_ops network_ethtool_ops =
+ {
+ .get_tx_csum = ethtool_op_get_tx_csum,
+ .set_tx_csum = ethtool_op_set_tx_csum,
+ };
+
+ /** Create a network device.
+ * @param handle device handle
+ * @param val return parameter for created device
+ * @return 0 on success, error code otherwise
+ */
+ static int create_netdev(int handle, struct xenbus_device *dev,
+ struct net_device **val)
+ {
+ int i, err = 0;
+ struct net_device *netdev = NULL;
+ struct netfront_info *np = NULL;
+
+ if ((netdev = alloc_etherdev(sizeof(struct netfront_info))) == NULL) {
+ printk(KERN_WARNING "%s> alloc_etherdev failed.\n",
+ __FUNCTION__);
+ err = -ENOMEM;
+ goto exit;
+ }
+
+ np = netdev_priv(netdev);
+ np->backend_state = BEST_CLOSED;
+ np->user_state = UST_CLOSED;
+ np->handle = handle;
+ np->xbdev = dev;
+
+ spin_lock_init(&np->tx_lock);
+ spin_lock_init(&np->rx_lock);
+
+ skb_queue_head_init(&np->rx_batch);
+ np->rx_target = RX_DFL_MIN_TARGET;
+ np->rx_min_target = RX_DFL_MIN_TARGET;
+ np->rx_max_target = RX_MAX_TARGET;

```

```

+
+ init_timer(&np->rx_refill_timer);
+ np->rx_refill_timer.data = (unsigned long)netdev;
+ np->rx_refill_timer.function = rx_refill_timeout;
+
+ /* Initialise {tx,rx}_skbs as a free chain containing every entry. */
+ for (i = 0; i <= NET_TX_RING_SIZE; i++) {
+ np->tx_skbs[i] = (void *)((unsigned long) i+1);
+ np->grant_tx_ref[i] = GRANT_INVALID_REF;
+ }
+
+ for (i = 0; i <= NET_RX_RING_SIZE; i++) {
+ np->rx_skbs[i] = (void *)((unsigned long) i+1);
+ np->grant_rx_ref[i] = GRANT_INVALID_REF;
+ }
+
+ /* A grant for every tx ring slot */
+ if (gnttab_alloc_grant_references(NET_TX_RING_SIZE,
+ &np->gref_tx_head) < 0) {
+ printk(KERN_ALERT "#### netfront can't alloc tx grant refs\n");
+ err = -ENOMEM;
+ goto exit;
+ }
+
+ /* A grant for every rx ring slot */
+ if (gnttab_alloc_grant_references(NET_RX_RING_SIZE,
+ &np->gref_rx_head) < 0) {
+ printk(KERN_ALERT "#### netfront can't alloc rx grant refs\n");
+ gnttab_free_grant_references(np->gref_tx_head);
+ err = -ENOMEM;
+ goto exit;
+ }
+
+ netdev->open = network_open;
+ netdev->hard_start_xmit = network_start_xmit;
+ netdev->stop = network_close;
+ netdev->get_stats = network_get_stats;
+ netdev->poll = netif_poll;
+ netdev->uninit = netif_uninit;
+ netdev->weight = 64;
+ netdev->features = NETIF_F_IP_CSUM;
+
+ SET_ETHTOOL_OPS(netdev, &network_ethtool_ops);
+ SET_MODULE_OWNER(netdev);
+ SET_NETDEV_DEV(netdev, &dev->dev);
+
+ if ((err = register_netdev(netdev)) != 0) {
+ printk(KERN_WARNING "%s> register_netdev err=%d\n",
+ __FUNCTION__, err);
+ goto exit_free_grefs;
+ }
+

```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ if ((err = xennet_proc_addif(netdev)) != 0) {
+ unregister_netdev(netdev);
+ goto exit_free_grefs;
+ }
+
+ np->netdev = netdev;
+
+ exit:
+ if (err != 0) {
+ if (netdev)
+ free_netdev(netdev);
+ } else if (val != NULL)
+ *val = netdev;
+ return err;
+
+ exit_free_grefs:
+ gnttab_free_grant_references(np->gref_tx_head);
+ gnttab_free_grant_references(np->gref_rx_head);
+ goto exit;
+}
+
+/*
+ * We use this notifier to send out a fake ARP reply to reset switches and
+ * router ARP caches when an IP interface is brought up on a VIF.
+ */
+static int
+inetdev_notify(struct notifier_block *this, unsigned long event, void *ptr)
+{
+ struct in_ifaddr *ifa = (struct in_ifaddr *)ptr;
+ struct net_device *dev = ifa->ifa_dev->dev;
+
+ /* UP event and is it one of our devices? */
+ if (event == NETDEV_UP && dev->open == network_open)
+ (void)send_fake_arp(dev);
+
+ return NOTIFY_DONE;
+}
+
+/* ** Close down ** */
+
+/*
+ * Handle the change of state of the backend to Closing. We must delete our
+ * device-layer structures now, to ensure that writes are flushed through to
+ * the backend. Once is this done, we can switch to Closed in
+ * acknowledgement.
+ */
+static void netfront_closing(struct xenbus_device *dev)
+{
+ struct netfront_info *info = dev->data;
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+
+ DPRINTK("netfront_closing: %s removed\n", dev->nodename);
+
+ close_netdev(info);
+
+ xenbus_switch_state(dev, XBT_NULL, XenbusStateClosed);
+}
+
+
+static int netfront_remove(struct xenbus_device *dev)
+{
+ struct netfront_info *info = dev->data;
+
+ DPRINTK("%s\n", dev->nodename);
+
+ netif_disconnect_backend(info);
+ free_netdev(info->netdev);
+
+ return 0;
+}
+
+
+static void close_netdev(struct netfront_info *info)
+{
+ spin_lock_irq(&info->netdev->xmit_lock);
+ netif_stop_queue(info->netdev);
+ spin_unlock_irq(&info->netdev->xmit_lock);
+
+ #if defined(CONFIG_PROC_FS) && defined(XEN_XENNET_PROC_INTERFACE)
+ xenet_proc_delif(info->netdev);
+ #endif
+
+ del_timer_sync(&info->rx_refill_timer);
+
+ unregister_netdev(info->netdev);
+}
+
+
+static void netif_disconnect_backend(struct netfront_info *info)
+{
+ /* Stop old i/f to prevent errors whilst we rebuild the state. */
+ spin_lock_irq(&info->tx_lock);
+ spin_lock(&info->rx_lock);
+ info->backend_state = BEST_DISCONNECTED;
+ spin_unlock(&info->rx_lock);
+ spin_unlock_irq(&info->tx_lock);
+
+ if (info->irq)
+ unbind_from_irqhandler(info->irq, info->netdev);
+ info->evtchn = info->irq = 0;
+
+}
```

[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ end_access(info->tx_ring_ref, info->tx.sring);
+ end_access(info->rx_ring_ref, info->rx.sring);
+ info->tx_ring_ref = GRANT_INVALID_REF;
+ info->rx_ring_ref = GRANT_INVALID_REF;
+ info->tx.sring = NULL;
+ info->rx.sring = NULL;
+}
+
+
+static void netif_free(struct netfront_info *info)
+{
+ close_netdev(info);
+ netif_disconnect_backend(info);
+ free_netdev(info->netdev);
+}
+
+
+static void end_access(int ref, void *page)
+{
+ if (ref != GRANT_INVALID_REF)
+ gnttab_end_foreign_access(ref, 0, (unsigned long)page);
+}
+
+
+/* ** Driver registration ** */
+
+
+static struct xenbus_device_id netfront_ids[] = {
+ { "vif" },
+ { "" }
+};
+
+
+static struct xenbus_driver netfront = {
+ .name = "vif",
+ .owner = THIS_MODULE,
+ .ids = netfront_ids,
+ .probe = netfront_probe,
+ .remove = netfront_remove,
+ .resume = netfront_resume,
+ .otherend_changed = backend_changed,
+};
+
+
+static struct notifier_block notifier_inetdev = {
+ .notifier_call = inetdev_notify,
+};
+
+static int __init netif_init(void)
+{
+ int err = 0;
```



```

+ case TARGET_CUR:
+ len = sprintf(page, "%d\n", np->rx_target);
+ break;
+ }
+
+ *eof = 1;
+ return len;
+}
+
+static int xennet_proc_write(
+ struct file *file, const char __user *buffer,
+ unsigned long count, void *data)
+{
+ struct net_device *dev =
+ (struct net_device *)((unsigned long)data & ~3UL);
+ struct netfront_info *np = netdev_priv(dev);
+ int which_target = (long)data & 3;
+ char string[64];
+ long target;
+
+ if (!capable(CAP_SYS_ADMIN))
+ return -EPERM;
+
+ if (count <= 1)
+ return -EBADMSG; /* runt */
+ if (count > sizeof(string))
+ return -EFBIG; /* too long */
+
+ if (copy_from_user(string, buffer, count))
+ return -EFAULT;
+ string[sizeof(string)-1] = '\0';
+
+ target = simple_strtol(string, NULL, 10);
+ if (target < RX_MIN_TARGET)
+ target = RX_MIN_TARGET;
+ if (target > RX_MAX_TARGET)
+ target = RX_MAX_TARGET;
+
+ spin_lock(&np->rx_lock);
+
+ switch (which_target) {
+ case TARGET_MIN:
+ if (target > np->rx_max_target)
+ np->rx_max_target = target;
+ np->rx_min_target = target;
+ if (target > np->rx_target)
+ np->rx_target = target;
+ break;
+ case TARGET_MAX:
+ if (target < np->rx_min_target)
+ np->rx_min_target = target;

```

```

+ np->rx_max_target = target;
+ if (target < np->rx_target)
+ np->rx_target = target;
+ break;
+ case TARGET_CUR:
+ break;
+ }
+
+ network_alloc_rx_buffers(dev);
+
+ spin_unlock(&np->rx_lock);
+
+ return count;
+}
+
+static int xennet_proc_init(void)
+{
+ if (proc_mkdir("xen/net", NULL) == NULL)
+ return -ENOMEM;
+ return 0;
+}
+
+static int xennet_proc_addif(struct net_device *dev)
+{
+ struct proc_dir_entry *dir, *min, *max, *cur;
+ char name[30];
+
+ sprintf(name, "xen/net/%s", dev->name);
+
+ dir = proc_mkdir(name, NULL);
+ if (!dir)
+ goto nomem;
+ dir->owner = THIS_MODULE;
+
+ min = create_proc_entry("rxbuf_min", 0644, dir);
+ max = create_proc_entry("rxbuf_max", 0644, dir);
+ cur = create_proc_entry("rxbuf_cur", 0444, dir);
+ if (!min || !max || !cur)
+ goto nomem;
+
+ min->read_proc = xennet_proc_read;
+ min->write_proc = xennet_proc_write;
+ min->data = (void *)((unsigned long)dev | TARGET_MIN);
+ min->owner = THIS_MODULE;
+
+ max->read_proc = xennet_proc_read;
+ max->write_proc = xennet_proc_write;
+ max->data = (void *)((unsigned long)dev | TARGET_MAX);
+ max->owner = THIS_MODULE;
+
+ cur->read_proc = xennet_proc_read;

```


[RFC PATCH 34/35] Add the Xen virtual network device driver.

```
+ * The above copyright notice and this permission notice shall be included in
+ * all copies or substantial portions of the Software.
+ *
+ * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
+ * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
+ * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
THE
+ * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
+ * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
+ * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
+ * DEALINGS IN THE SOFTWARE.
+ */
+
+#ifndef _ASM_XEN_NET_DRIVER_UTIL_H
+#define _ASM_XEN_NET_DRIVER_UTIL_H
+
+#include <xen/xenbus.h>
+
+/**
+ * Read the 'mac' node at the given device's node in the store, and parse that
+ * as colon-separated octets, placing result the given mac array. mac must be
+ * a preallocated array of length ETH_ALEN (as declared in linux/if_ether.h).
+ * Return 0 on success, or -errno on error.
+ */
+int xen_net_read_mac(struct xenbus_device *dev, u8 mac[]);
+
+#endif /* _ASM_XEN_NET_DRIVER_UTIL_H */
```

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
the body of a message to majordomo@xxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>