sk_buff

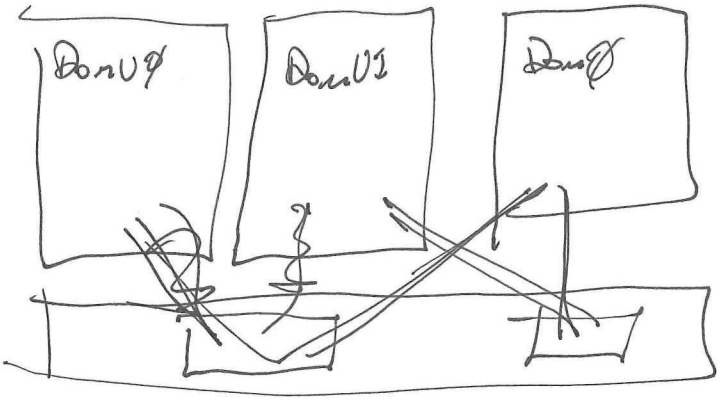
The `sk_buff` data structure is used to describe network data as it moves between the layers of protocol.

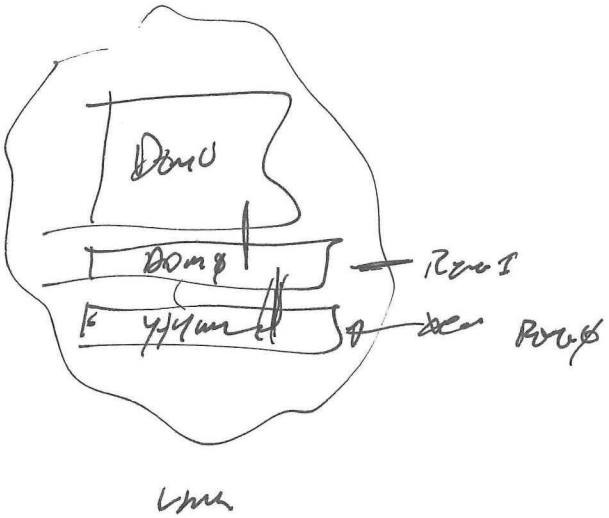
```
struct sk_buff
{
  struct sk_buff      *next;       /* Next buffer in list                  */
  struct sk_buff      *prev;       /* Previous buffer in list              */
  struct sk_buff_head *list;       /* List we are on                       */
  int                 magic_debug_cookie;
  struct sk_buff      *link3;      /* Link for IP protocol level buffer chains */
  struct sock         *sk;         /* Socket we are owned by               */
  unsigned long       when;        /* used to compute rtt's                */
  struct timeval      stamp;       /* Time we arrived                      */
  struct device       *dev;        /* Device we arrived on/are leaving by  */
  union
  {
      struct tcphdr   *th;
      struct ethhdr   *eth;
      struct iphdr    *iph;
      struct udphdr   *uh;
      unsigned char   *raw;
      /* for passing file handles in a unix domain socket */
      void            *filp;
  } h;

  union
  {
      /* As yet incomplete physical layer views */
      unsigned char   *raw;
      struct ethhdr   *ethernet;
  } mac;

  struct iphdr        *ip_hdr;     /* For IPPROTO_RAW                      */
  unsigned long       len;         /* Length of actual data                */
  unsigned long       csum;        /* Checksum                             */
  __u32               saddr;       /* IP source address                    */
  __u32               daddr;       /* IP target address                    */
  __u32               raddr;       /* IP next hop address                  */
  __u32               seq;         /* TCP sequence number                  */
  __u32               end_seq;     /* seq [+ fin] [+ syn] + datalen        */
  __u32               ack_seq;     /* TCP ack sequence number              */
  unsigned char       proto_priv[16];
  volatile char       acked,       /* Are we acked ?                       */
                      used,        /* Are we in use ?                      */
                      free,        /* How to free this buffer              */
                      arp;         /* Has IP/ARP resolution finished       */
  unsigned char       tries,       /* Times tried                          */
                      lock,        /* Are we locked ?                      */
                      localroute,  /* Local routing asserted for this frame */
                      pkt_type,    /* Packet class                         */
                      pkt_bridged, /* Tracker for bridging                 */
                      ip_summed;   /* Driver fed us an IP checksum         */
#define PACKET_HOST         0       /* To us                               */
#define PACKET_BROADCAST    1       /* To all                              */
#define PACKET_MULTICAST    2       /* To group                            */
#define PACKET_OTHERHOST    3       /* To someone else                     */
  unsigned short      users;       /* User count - see datagram.c,tcp.c    */
  unsigned short      protocol;    /* Packet protocol from driver.         */
  unsigned int        truesize;    /* Buffer size                          */
  atomic_t            count;       /* reference count                      */
  struct sk_buff      *data_skb;   /* Link to the actual data skb          */
  unsigned char       *head;       /* Head of buffer                       */
  unsigned char       *data;       /* Data head pointer                    */
  unsigned char       *tail;       /* Tail pointer                         */
  unsigned char       *end;        /* End pointer                          */
  void                (*destructor)(struct sk_buff *); /* Destruct function */
  __u16               redirport;   /* Redirect port                        */
};
```

timer_list

`timer_list` data structure's are used to implement real time timers for processes.

```
struct timer_list {
  struct timer_list *next;
  struct timer_list *prev;
  unsigned long expires;
  unsigned long data;
  void (*function)(unsigned long);
};
```





```
[root@ ~]# xenstore-ls /local/domain/1
vm = "/vm/69094090-7a72-59d4-0c47-e89466a4cd97"
vss = "/vss/69094090-7a72-59d4-0c47-e89466a4cd97"
name = ""
cpu = ""
 0 = ""
  availability = "online"
 1 = ""
  availability = "online"
memory = ""
 static-max = "2097152"
 target = "2097152"
device = ""
 vbd = ""
  51760 = ""
   backend = "/local/domain/0/backend/vbd/1/51760"
   protocol = "x86_32-abi"
   state = "6"
   backend-id = "0"
   device-type = "cdrom"
   virtual-device = "51760"
   ring-ref = "8"
   event-channel = "9"
  51712 = ""
   backend = "/local/domain/0/backend/tap/1/51712"
   protocol = "x86_32-abi"
   state = "4"
   backend-id = "0"
   device-type = "disk"
   virtual-device = "51712"
   ring-ref = "9"
   event-channel = "10"
 vif = ""
  0 = ""
   backend = "/local/domain/0/backend/vif/1/0"
   backend-id = "0"
   state = "4"
   handle = "0"
   mac = "da:e4:af:02:2a:d6"
   protocol = "x86_32-abi"
   tx-ring-ref = "2304"
   rx-ring-ref = "2305"
   event-channel = "11"
   request-rx-copy = "1"
   feature-rx-notify = "1"
   feature-sg = "1"
```

```
      feature-gso-tcpv4 = "1"
    1 = ""
      backend = "/local/domain/0/backend/vif/1/1"
      backend-id = "0"
      state = "4"
      handle = "1"
      mac = "06:7a:46:1e:83:33"
      protocol = "x86_32-abi"
      tx-ring-ref = "2306"
      rx-ring-ref = "2307"
      event-channel = "12"
      request-rx-copy = "1"
      feature-rx-notify = "1"
      feature-sg = "1"
      feature-gso-tcpv4 = "1"
    2 = ""
      backend = "/local/domain/0/backend/vif/1/2"
      backend-id = "0"
      state = "4"
      handle = "2"
      mac = "de:24:a9:b3:e9:3a"
      protocol = "x86_32-abi"
      tx-ring-ref = "2308"
      rx-ring-ref = "2309"
      event-channel = "13"
      request-rx-copy = "1"
      feature-rx-notify = "1"
      feature-sg = "1"
      feature-gso-tcpv4 = "1"
    3 = ""
      backend = "/local/domain/0/backend/vif/1/3"
      backend-id = "0"
      state = "4"
      handle = "3"
      mac = "ce:e9:2d:95:dd:04"
      protocol = "x86_32-abi"
      tx-ring-ref = "2310"
      rx-ring-ref = "2311"
      event-channel = "14"
      request-rx-copy = "1"
      feature-rx-notify = "1"
      feature-sg = "1"
      feature-gso-tcpv4 = "1"
  error = ""
  drivers = ""
  control = ""
```

```
  platform-feature-multiprocessor-suspend = "1"
 attr = ""
 data = ""
 messages = ""
 platform = ""
  nx = "false"
  acpi = "true"
  apic = "true"
  pae = "true"
  viridian = "true"
 domid = "1"
 store = ""
  port = "1"
  ring-ref = "9432981"
 serial = ""
  0 = ""
   limit = "65536"
   tty = "/dev/pts/1"
   vncterm-pid = "17663"
   vnc-port = "5901"
 console = ""
  port = "2"
  ring-ref = "9432980"
  tty = "/dev/pts/1"
```

```
/*****************************************************************************
 * drivers/xen/netback/accel.c
 *
 * Interface between backend virtual network device and accelerated plugin.
 *
 * Copyright (C) 2007 Solarflare Communications, Inc
 *
 * 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/list.h>
#include <asm/atomic.h>
#include <xen/xenbus.h>
#include <linux/mutex.h>

#include "common.h"

#if 0
#undef DPRINTK
#define DPRINTK(fmt, args...)                                   \
    printk("netback/accel (%s:%d) " fmt ".\n", __FUNCTION__, __LINE__, ##args)
#endif

/*
 * A list of available netback accelerator plugin modules (each list
 * entry is of type struct netback_accelerator)
 */
static struct list_head accelerators_list;
/* Lock used to protect access to accelerators_list */
DEFINE_MUTEX(accelerators_mutex);

/*
 * Compare a backend to an accelerator, and decide if they are
 * compatible (i.e. if the accelerator should be used by the
 * backend)
```

```
 */
static int match_accelerator(struct xenbus_device *xendev,
                             struct backend_info *be,
                             struct netback_accelerator *accelerator)
{
    int rc = 0;
    char *eth_name = xenbus_read(XBT_NIL, xendev->nodename, "accel", NULL);

    if (IS_ERR(eth_name)) {
        int dom, vif;
        char *nodename;

        /* Try alternate configuration location */
        sscanf(xendev->nodename, "backend/vif/%d/%d", &dom, &vif);
        nodename = kasprintf(GFP_NOIO | __GFP_HIGH,
                             "/local/domain/%d/vm-data/device/vif/%d",
                             dom, vif);
        if (!nodename)
            return 0;

        eth_name = xenbus_read(XBT_NIL, nodename, "accel", NULL);

        kfree(nodename);

        if (IS_ERR(eth_name)) {
            /* Probably means not present */
            DPRINTK("%s: no match due to xenbus_read accel err %ld\n",
                __FUNCTION__, PTR_ERR(eth_name));
            return 0;
        }
    }
    if (!strcmp(eth_name, accelerator->eth_name))
        rc = 1;
    kfree(eth_name);
    return rc;
}


static void do_probe(struct backend_info *be,
             struct netback_accelerator *accelerator,
             struct xenbus_device *xendev)
{
    be->accelerator = accelerator;
    atomic_inc(&be->accelerator->use_count);
    if (be->accelerator->hooks->probe(xendev) != 0) {
        atomic_dec(&be->accelerator->use_count);
        module_put(be->accelerator->hooks->owner);
        be->accelerator = NULL;
    }
}


/*
 * Notify suitable backends that a new accelerator is available and
 * connected.  This will also notify the accelerator plugin module
 * that it is being used for a device through the probe hook.
 */
```

```
static int netback_accelerator_probe_backend(struct device *dev, void *arg)
{
    struct netback_accelerator *accelerator =
        (struct netback_accelerator *)arg;
    struct xenbus_device *xendev = to_xenbus_device(dev);

    if (!strcmp("vif", xendev->devicetype)) {
        struct backend_info *be = xendev->dev.driver_data;

        if (match_accelerator(xendev, be, accelerator) &&
            try_module_get(accelerator->hooks->owner)) {
            do_probe(be, accelerator, xendev);
        }
    }
    return 0;
}


/*
 * Notify suitable backends that an accelerator is unavailable.
 */
static int netback_accelerator_remove_backend(struct device *dev, void *arg)
{
    struct xenbus_device *xendev = to_xenbus_device(dev);
    struct netback_accelerator *accelerator =
        (struct netback_accelerator *)arg;

    if (!strcmp("vif", xendev->devicetype)) {
        struct backend_info *be = xendev->dev.driver_data;

        if (be->accelerator == accelerator) {
            be->accelerator->hooks->remove(xendev);
            atomic_dec(&be->accelerator->use_count);
            module_put(be->accelerator->hooks->owner);
            be->accelerator = NULL;
        }
    }
    return 0;
}



/*
 * Entry point for an netback accelerator plugin module.  Called to
 * advertise its presence, and connect to any suitable backends.
 */
int netback_connect_accelerator(unsigned version, int id, const char *eth_name,
                struct netback_accel_hooks *hooks)
{
    struct netback_accelerator *new_accelerator;
    unsigned eth_name_len;

    if (version != NETBACK_ACCEL_VERSION) {
        if (version > NETBACK_ACCEL_VERSION) {
            /* Caller has higher version number, leave it
               up to them to decide whether to continue.
               They can recall with a lower number if
```

```
                they're happy to be compatible with us */
            return NETBACK_ACCEL_VERSION;
        } else {
            /* We have a more recent version than caller.
               Currently reject, but may in future be able
               to be backwardly compatible */
            return -EPROTO;
        }
    }

    new_accelerator =
        kmalloc(sizeof(struct netback_accelerator), GFP_KERNEL);
    if (!new_accelerator) {
        DPRINTK("%s: failed to allocate memory for accelerator\n",
            __FUNCTION__);
        return -ENOMEM;
    }

    new_accelerator->id = id;

    eth_name_len = strlen(eth_name)+1;
    new_accelerator->eth_name = kmalloc(eth_name_len, GFP_KERNEL);
    if (!new_accelerator->eth_name) {
        DPRINTK("%s: failed to allocate memory for eth_name string\n",
            __FUNCTION__);
        kfree(new_accelerator);
        return -ENOMEM;
    }
    strlcpy(new_accelerator->eth_name, eth_name, eth_name_len);

    new_accelerator->hooks = hooks;

    atomic_set(&new_accelerator->use_count, 0);

    mutex_lock(&accelerators_mutex);
    list_add(&new_accelerator->link, &accelerators_list);

    /* tell existing backends about new plugin */
    xenbus_for_each_backend(new_accelerator,
                netback_accelerator_probe_backend);

    mutex_unlock(&accelerators_mutex);

    return 0;

}
EXPORT_SYMBOL_GPL(netback_connect_accelerator);


/*
 * Disconnect an accelerator plugin module that has previously been
 * connected.
 */
void netback_disconnect_accelerator(int id, const char *eth_name)
{
    struct netback_accelerator *accelerator, *next;
```

```
    mutex_lock(&accelerators_mutex);
    list_for_each_entry_safe(accelerator, next, &accelerators_list, link) {
        if (!strcmp(eth_name, accelerator->eth_name)) {
            xenbus_for_each_backend
                (accelerator, netback_accelerator_remove_backend);
            BUG_ON(atomic_read(&accelerator->use_count) != 0);
            list_del(&accelerator->link);
            kfree(accelerator->eth_name);
            kfree(accelerator);
            break;
        }
    }
    mutex_unlock(&accelerators_mutex);
}
EXPORT_SYMBOL_GPL(netback_disconnect_accelerator);


void netback_probe_accelerators(struct backend_info *be,
                struct xenbus_device *dev)
{
    struct netback_accelerator *accelerator;

    /*
     * Check list of accelerators to see if any is suitable, and
     * use it if it is.
     */
    mutex_lock(&accelerators_mutex);
    list_for_each_entry(accelerator, &accelerators_list, link) {
        if (match_accelerator(dev, be, accelerator) &&
            try_module_get(accelerator->hooks->owner)) {
            do_probe(be, accelerator, dev);
            break;
        }
    }
    mutex_unlock(&accelerators_mutex);
}


void netback_remove_accelerators(struct backend_info *be,
                 struct xenbus_device *dev)
{
    mutex_lock(&accelerators_mutex);
    /* Notify the accelerator (if any) of this device's removal */
    if (be->accelerator != NULL) {
        be->accelerator->hooks->remove(dev);
        atomic_dec(&be->accelerator->use_count);
        module_put(be->accelerator->hooks->owner);
        be->accelerator = NULL;
    }
    mutex_unlock(&accelerators_mutex);
}


void netif_accel_init(void)
{
    INIT_LIST_HEAD(&accelerators_list);
}
```

```
/******************************************************************************
 * arch/xen/drivers/netif/backend/common.h
 *
 * 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.
 */

#ifndef __NETIF__BACKEND__COMMON_H__
#define __NETIF__BACKEND__COMMON_H__

#include <linux/version.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/slab.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <linux/netdevice.h>
#include <linux/etherdevice.h>
#include <linux/wait.h>
#include <xen/evtchn.h>
#include <xen/interface/io/netif.h>
#include <asm/io.h>
#include <asm/pgalloc.h>
#include <xen/interface/grant_table.h>
#include <xen/gnttab.h>
#include <xen/driver_util.h>
#include <xen/xenbus.h>

#define DPRINTK(_f, _a...)                  \
    pr_debug("(file=%s, line=%d) " _f,  \
           __FILE__ , __LINE__ , ## _a )
#define IPRINTK(fmt, args...)                \
    printk(KERN_INFO "xen_net: " fmt, ##args)
#define WPRINTK(fmt, args...)                \
    printk(KERN_WARNING "xen_net: " fmt, ##args)
```

```
struct backend_info;

struct xen_netif {
    /* Unique identifier for this interface. */
    domid_t          domid;
    unsigned int     handle;

    u8               fe_dev_addr[6];

    /* Physical parameters of the comms window. */
    grant_handle_t   tx_shmem_handle;
    grant_ref_t      tx_shmem_ref;
    grant_handle_t   rx_shmem_handle;
    grant_ref_t      rx_shmem_ref;
    unsigned int     irq;

    /* The shared rings and indexes. */
    struct xen_netif_tx_back_ring tx;
    struct xen_netif_rx_back_ring rx;
    struct vm_struct *tx_comms_area;
    struct vm_struct *rx_comms_area;

    /* Set of features that can be turned on in dev->features. */
    int features;

    /* Internal feature information. */
    u8 can_queue:1; /* can queue packets for receiver? */
    u8 gso_prefix:1;    /* use a prefix segment for GSO information */
    u8 copying_receiver:1;  /* copy packets to receiver? */

    /* Allow netif_be_start_xmit() to peek ahead in the rx request
     * ring.  This is a prediction of what rx_req_cons will be once
     * all queued skbs are put on the ring. */
    RING_IDX rx_req_cons_peek;

    /* Transmit shaping: allow 'credit_bytes' every 'credit_usec'. */
    unsigned long   credit_bytes;
    unsigned long   credit_usec;
    unsigned long   remaining_credit;
    struct timer_list credit_timeout;

    /* Enforce draining of the transmit queue. */
    struct timer_list tx_queue_timeout;

    /* Statistics */
    int nr_copied_skbs;

    /* Miscellaneous private stuff. */
    struct list_head list;  /* scheduling list */
    atomic_t         refcnt;
    struct net_device *dev;
    struct net_device_stats stats;
    struct backend_info *be;

    unsigned int carrier;

    wait_queue_head_t waiting_to_free;
```

```
};

/*
 * Implement our own carrier flag: the network stack's version causes delays
 * when the carrier is re-enabled (in particular, dev_activate() may not
 * immediately be called, which can cause packet loss; also the etherbridge
 * can be rather lazy in activating its port).
 */
#define netback_carrier_on(netif)    ((netif)->carrier = 1)
#define netback_carrier_off(netif)   ((netif)->carrier = 0)
#define netback_carrier_ok(netif)    ((netif)->carrier)

enum {
    NETBK_DONT_COPY_SKB,
    NETBK_DELAYED_COPY_SKB,
    NETBK_ALWAYS_COPY_SKB,
};

extern int netbk_copy_skb_mode;

/* Function pointers into netback accelerator plugin modules */
struct netback_accel_hooks {
    struct module *owner;
    int  (*probe)(struct xenbus_device *dev);
    int (*remove)(struct xenbus_device *dev);
};

/* Structure to track the state of a netback accelerator plugin */
struct netback_accelerator {
    struct list_head link;
    int id;
    char *eth_name;
    atomic_t use_count;
    struct netback_accel_hooks *hooks;
};

struct backend_info {
    struct xenbus_device *dev;
    struct xen_netif *netif;
    enum xenbus_state frontend_state;
    struct xenbus_watch hotplug_status_watch;
    struct xenbus_watch csum_offload_watch;
    int have_hotplug_status_watch:1;
    int have_csum_offload:1;
    int have_csum_offload_watch:1;

    /* State relating to the netback accelerator */
    void *netback_accel_priv;
    /* The accelerator that this backend is currently using */
    struct netback_accelerator *accelerator;
};

#define NETBACK_ACCEL_VERSION 0x00010001

/*
 * Connect an accelerator plugin module to netback.  Returns zero on
 * success, < 0 on error, > 0 (with highest version number supported)
```

```
 * if version mismatch.
 */
extern int netback_connect_accelerator(unsigned version,
                       int id, const char *eth_name,
                       struct netback_accel_hooks *hooks);
/* Disconnect a previously connected accelerator plugin module */
extern void netback_disconnect_accelerator(int id, const char *eth_name);


extern
void netback_probe_accelerators(struct backend_info *be,
                struct xenbus_device *dev);
extern
void netback_remove_accelerators(struct backend_info *be,
                 struct xenbus_device *dev);
extern
void netif_accel_init(void);


#define NET_TX_RING_SIZE __RING_SIZE((struct xen_netif_tx_sring *)0, PAGE_SIZE)
#define NET_RX_RING_SIZE __RING_SIZE((struct xen_netif_rx_sring *)0, PAGE_SIZE)

void netif_disconnect(struct xen_netif *netif);

struct xen_netif *netif_alloc(struct device *parent, domid_t domid, unsigned int handle)
int netif_map(struct xen_netif *netif, unsigned long tx_ring_ref,
          unsigned long rx_ring_ref, unsigned int evtchn);

static inline void netif_get(struct xen_netif *netif)
{
    atomic_inc(&netif->refcnt);
}

static inline void  netif_put(struct xen_netif *netif)
{
    if (atomic_dec_and_test(&netif->refcnt))
        wake_up(&netif->waiting_to_free);
}

void netif_xenbus_init(void);

#define netif_schedulable(netif)                \
    (netif_running((netif)->dev) && netback_carrier_ok(netif))

void netif_schedule_work(struct xen_netif *netif);
void netif_deschedule_work(struct xen_netif *netif);

int netif_be_start_xmit(struct sk_buff *skb, struct net_device *dev);
struct net_device_stats *netif_be_get_stats(struct net_device *dev);
irqreturn_t netif_be_int(int irq, void *dev_id);

void netif_set_tx_csum(struct backend_info *, u32);

static inline int netbk_can_queue(struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);
    return netif->can_queue;
```

```
}

static inline int netbk_can_sg(struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);
    return netif->features & NETIF_F_SG;
}

#endif /* __NETIF__BACKEND__COMMON_H__ */
```

```
/*  Xenbus code for netif backend
    Copyright (C) 2005 Rusty Russell <rusty@rustcorp.com.au>
    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 as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
*/

#include <stdarg.h>
#include <linux/module.h>
#include <xen/xenbus.h>
#include "common.h"

#if 0
#undef DPRINTK
#define DPRINTK(fmt, args...) \
    printk("netback/xenbus (%s:%d) " fmt ".\n", __FUNCTION__, __LINE__, ##args)
#endif


static int connect_rings(struct backend_info *);
static void connect(struct backend_info *);
static void backend_create_netif(struct backend_info *be);
static void unregister_hotplug_status_watch(struct backend_info *be);
static int register_csum_offload_watch(struct backend_info *be);
static void unregister_csum_offload_watch(struct backend_info *be);

static int netback_remove(struct xenbus_device *dev)
{
    struct backend_info *be = dev->dev.driver_data;

    netback_remove_accelerators(be, dev);

    unregister_csum_offload_watch(be);
    unregister_hotplug_status_watch(be);
    if (be->netif) {
        kobject_uevent(&dev->dev.kobj, KOBJ_OFFLINE);
        xenbus_rm(XBT_NIL, dev->nodename, "hotplug-status");
        netif_disconnect(be->netif);
        be->netif = NULL;
    }
    kfree(be);
    dev->dev.driver_data = NULL;
    return 0;
}
```

```
/**
 * Entry point to this code when a new device is created.  Allocate the basic
 * structures and switch to InitWait.
 */
static int netback_probe(struct xenbus_device *dev,
             const struct xenbus_device_id *id)
{
    const char *message;
    struct xenbus_transaction xbt;
    int err;
    int sg;
    struct backend_info *be = kzalloc(sizeof(struct backend_info),
                      GFP_KERNEL);
    if (!be) {
        xenbus_dev_fatal(dev, -ENOMEM,
                 "allocating backend structure");
        return -ENOMEM;
    }

    be->dev = dev;
    dev->dev.driver_data = be;

    be->have_csum_offload = 1;

    sg = 1;
    if (netbk_copy_skb_mode == NETBK_ALWAYS_COPY_SKB)
        sg = 0;

    do {
        err = xenbus_transaction_start(&xbt);
        if (err) {
            xenbus_dev_fatal(dev, err, "starting transaction");
            goto fail;
        }

        err = xenbus_printf(xbt, dev->nodename, "feature-sg", "%d", sg);
        if (err) {
            message = "writing feature-sg";
            goto abort_transaction;
        }

        err = xenbus_printf(xbt, dev->nodename, "feature-gso-tcpv4",
                    "%d", sg);
        if (err) {
            message = "writing feature-gso-tcpv4";
            goto abort_transaction;
        }

        /* We support rx-copy path. */
        err = xenbus_printf(xbt, dev->nodename,
                    "feature-rx-copy", "%d", 1);
        if (err) {
            message = "writing feature-rx-copy";
            goto abort_transaction;
        }
```

```
        /* We can cope with transmit checksum offload packets
           in which the TCP and IP headers are in separate
           fragments. */
        err = xenbus_printf(xbt, dev->nodename,
                    "feature-tx-csum-split-header", "%d", 1);
        if (err) {
            message = "writing feature-tx-csum-split-header";
            goto abort_transaction;
        }

        /*
         * We don't support rx-flip path (except old guests who don't
         * grok this feature flag).
         */
        err = xenbus_printf(xbt, dev->nodename,
                    "feature-rx-flip", "%d", 0);
        if (err) {
            message = "writing feature-rx-flip";
            goto abort_transaction;
        }

        err = xenbus_transaction_end(xbt, 0);
    } while (err == -EAGAIN);

    if (err) {
        xenbus_dev_fatal(dev, err, "completing transaction");
        goto fail;
    }

    netback_probe_accelerators(be, dev);

    err = xenbus_switch_state(dev, XenbusStateInitWait);
    if (err)
        goto fail;

    /* This kicks hotplug scripts, so do it immediately. */
    backend_create_netif(be);

    return 0;

abort_transaction:
    xenbus_transaction_end(xbt, 1);
    xenbus_dev_fatal(dev, err, "%s", message);
fail:
    DPRINTK("failed");
    netback_remove(dev);
    return err;
}


/**
 * Handle the creation of the hotplug script environment.  We add the script
 * and vif variables to the environment, for the benefit of the vif-* hotplug
 * scripts.
 */
static int netback_uevent(struct xenbus_device *xdev, struct kobj_uevent_env *env)
{
```

```
    struct backend_info *be = xdev->dev.driver_data;
    struct xen_netif *netif = be->netif;
    char *val;

    DPRINTK("netback_uevent");

    val = xenbus_read(XBT_NIL, xdev->nodename, "script", NULL);
    if (IS_ERR(val)) {
        int err = PTR_ERR(val);
        xenbus_dev_fatal(xdev, err, "reading script");
        return err;
    }
    else {
        add_uevent_var(env, "script=%s", val);
        kfree(val);
    }

    add_uevent_var(env, "vif=%s", netif->dev->name);

    return 0;
}


static void backend_create_netif(struct backend_info *be)
{
    int err;
    long handle;
    struct xenbus_device *dev = be->dev;

    if (be->netif != NULL)
        return;

    err = xenbus_scanf(XBT_NIL, dev->nodename, "handle", "%li", &handle);
    if (err != 1) {
        xenbus_dev_fatal(dev, err, "reading handle");
        return;
    }

    be->netif = netif_alloc(&dev->dev, dev->otherend_id, handle);
    if (IS_ERR(be->netif)) {
        err = PTR_ERR(be->netif);
        be->netif = NULL;
        xenbus_dev_fatal(dev, err, "creating interface");
        return;
    }
    be->netif->be = be;

    kobject_uevent(&dev->dev.kobj, KOBJ_ONLINE);
}


static void disconnect_backend(struct xenbus_device *dev)
{
    struct backend_info *be = dev->dev.driver_data;

    if (be->netif) {
        kobject_uevent(&dev->dev.kobj, KOBJ_OFFLINE);
```

```
        unregister_csum_offload_watch(be);
        xenbus_rm(XBT_NIL, dev->nodename, "hotplug-status");
        netif_disconnect(be->netif);
        be->netif = NULL;
    }
}

/**
 * Callback received when the frontend's state changes.
 */
static void frontend_changed(struct xenbus_device *dev,
                             enum xenbus_state frontend_state)
{
    struct backend_info *be = dev->dev.driver_data;

    DPRINTK("%s", xenbus_strstate(frontend_state));

    be->frontend_state = frontend_state;

    switch (frontend_state) {
    case XenbusStateInitialising:
        if (dev->state == XenbusStateClosed) {
            printk(KERN_INFO "%s: %s: prepare for reconnect\n",
                   __FUNCTION__, dev->nodename);
            xenbus_switch_state(dev, XenbusStateInitWait);
        }
        break;

    case XenbusStateInitialised:
        break;

    case XenbusStateConnected:
        if (dev->state == XenbusStateConnected)
            break;
        backend_create_netif(be);
        if (be->netif)
            connect(be);
        break;

    case XenbusStateClosing:
        xenbus_switch_state(dev, XenbusStateClosing);
        break;

    case XenbusStateClosed:
        disconnect_backend(dev);
        xenbus_switch_state(dev, XenbusStateClosed);
        if (xenbus_dev_is_online(dev))
            break;
        /* fall through if not online */
    case XenbusStateUnknown:
        device_unregister(&dev->dev);
        break;

    default:
        xenbus_dev_fatal(dev, -EINVAL, "saw state %d at frontend",
                         frontend_state);
        break;
```

```
    }
}


static void xen_net_read_rate(struct xenbus_device *dev,
                              unsigned long *bytes, unsigned long *usec)
{
    char *s, *e;
    unsigned long b, u;
    char *ratestr;

    /* Default to unlimited bandwidth. */
    *bytes = ~0UL;
    *usec = 0;

    ratestr = xenbus_read(XBT_NIL, dev->nodename, "rate", NULL);
    if (IS_ERR(ratestr))
        return;

    s = ratestr;
    b = simple_strtoul(s, &e, 10);
    if ((s == e) || (*e != ','))
        goto fail;

    s = e + 1;
    u = simple_strtoul(s, &e, 10);
    if ((s == e) || (*e != '\0'))
        goto fail;

    *bytes = b;
    *usec = u;

    kfree(ratestr);
    return;

 fail:
    WPRINTK("Failed to parse network rate limit. Traffic unlimited.\n");
    kfree(ratestr);
}

static int xen_net_read_mac(struct xenbus_device *dev, u8 mac[])
{
    char *s, *e, *macstr;
    int i;

    macstr = s = xenbus_read(XBT_NIL, dev->nodename, "mac", NULL);
    if (IS_ERR(macstr))
        return PTR_ERR(macstr);

    for (i = 0; i < ETH_ALEN; i++) {
        mac[i] = simple_strtoul(s, &e, 16);
        if ((s == e) || (*e != ((i == ETH_ALEN-1) ? '\0' : ':'))) {
            kfree(macstr);
            return -ENOENT;
        }
        s = e+1;
    }
```

```
    kfree(macstr);
    return 0;
}

static void unregister_hotplug_status_watch(struct backend_info *be)
{
    if (be->have_hotplug_status_watch) {
        unregister_xenbus_watch(&be->hotplug_status_watch);
        kfree(be->hotplug_status_watch.node);
    }
    be->have_hotplug_status_watch = 0;
}

static void hotplug_status_changed(struct xenbus_watch *watch,
                   const char **vec,
                   unsigned int vec_size)
{
    struct backend_info *be = container_of(watch,
                           struct backend_info,
                           hotplug_status_watch);
    char *str;
    unsigned int len;

    str = xenbus_read(XBT_NIL, be->dev->nodename, "hotplug-status", &len);
    if ( IS_ERR(str) )
        return;
    if ( len == sizeof("connected")-1 && !memcmp(str, "connected", len) ) {
        xenbus_switch_state(be->dev, XenbusStateConnected);
        /* Not interested in this watch anymore. */
        unregister_hotplug_status_watch(be);
    }
    kfree(str);
}

static void connect(struct backend_info *be)
{
    int err;
    struct xenbus_device *dev = be->dev;

    err = connect_rings(be);
    if (err)
        return;

    err = xen_net_read_mac(dev, be->netif->fe_dev_addr);
    if (err) {
        xenbus_dev_fatal(dev, err, "parsing %s/mac", dev->nodename);
        return;
    }

    xen_net_read_rate(dev, &be->netif->credit_bytes,
              &be->netif->credit_usec);
    be->netif->remaining_credit = be->netif->credit_bytes;

    unregister_hotplug_status_watch(be);
    err = xenbus_watch_path2(dev, dev->nodename, "hotplug-status",
                 &be->hotplug_status_watch,
```

```
                hotplug_status_changed);
    if (err) {
        /* Switch now, since we can't do a watch. */
        xenbus_switch_state(dev, XenbusStateConnected);
    } else {
        be->have_hotplug_status_watch = 1;
    }

    netif_wake_queue(be->netif->dev);
}

static void feature_csum_offload_changed(struct xenbus_watch *watch,
                    const char **vec,
                    unsigned int vec_size)
{
    int val;
    struct backend_info *be = container_of(watch,
                        struct backend_info,
                        csum_offload_watch);

    if (xenbus_scanf(XBT_NIL, be->dev->otherend, "feature-no-csum-offload",
             "%d", &val) < 0)
        val = 0;

    if (val) {
        be->netif->features &= ~NETIF_F_IP_CSUM;
        be->netif->dev->features &= ~NETIF_F_IP_CSUM;
    } else {
        be->netif->features |= NETIF_F_IP_CSUM;
        be->netif->dev->features |= NETIF_F_IP_CSUM;
    }
}
static int register_csum_offload_watch(struct backend_info *be)
{
    struct xenbus_device *dev = be->dev;
    int err;

    if (!be->have_csum_offload || be->have_csum_offload_watch)
        return 0;

    err = xenbus_watch_path2(dev, dev->otherend, "feature-no-csum-offload",
                &be->csum_offload_watch,
                feature_csum_offload_changed);
    if (err) {
        xenbus_dev_fatal(dev, err,
                "watching %s/feature-no-csum-offload",
                dev->otherend);
        return err;
    }
    be->have_csum_offload_watch = 1;
    return 0;

}

static void unregister_csum_offload_watch(struct backend_info *be)
{
    if (be->have_csum_offload_watch) {
```

```
        unregister_xenbus_watch(&be->csum_offload_watch);
        kfree(be->csum_offload_watch.node);
    }
    be->have_csum_offload_watch = 0;
}

void netif_set_tx_csum(struct backend_info *be, u32 data)
{
    be->have_csum_offload = data;
    if (data)
        register_csum_offload_watch(be);
    else
        unregister_csum_offload_watch(be);
}

static int connect_rings(struct backend_info *be)
{
    struct xenbus_device *dev = be->dev;
    unsigned long tx_ring_ref, rx_ring_ref;
    unsigned int evtchn, rx_copy;
    int err;
    int val;

    DPRINTK("");

    err = xenbus_gather(XBT_NIL, dev->otherend,
                "tx-ring-ref", "%lu", &tx_ring_ref,
                "rx-ring-ref", "%lu", &rx_ring_ref,
                "event-channel", "%u", &evtchn, NULL);
    if (err) {
        xenbus_dev_fatal(dev, err,
                 "reading %s/ring-ref and event-channel",
                 dev->otherend);
        return err;
    }

    err = xenbus_scanf(XBT_NIL, dev->otherend, "request-rx-copy", "%u",
               &rx_copy);
    if (err == -ENOENT) {
        err = 0;
        rx_copy = 0;
    }
    if (err < 0) {
        xenbus_dev_fatal(dev, err, "reading %s/request-rx-copy",
                 dev->otherend);
        return err;
    }
    be->netif->copying_receiver = !!rx_copy;

    if (be->netif->dev->tx_queue_len != 0) {
        if (xenbus_scanf(XBT_NIL, dev->otherend,
                 "feature-rx-notify", "%d", &val) < 0)
            val = 0;
        if (val)
            be->netif->can_queue = 1;
        else
            /* Must be non-zero for pfifo_fast to work. */
```

```
            be->netif->dev->tx_queue_len = 1;
    }

    if (xenbus_scanf(XBT_NIL, dev->otherend, "feature-sg", "%d", &val) < 0)
        val = 0;
    if (!val) {
        be->netif->features &= ~NETIF_F_SG;
        be->netif->dev->features &= ~NETIF_F_SG;
        if (be->netif->dev->mtu > ETH_DATA_LEN)
            be->netif->dev->mtu = ETH_DATA_LEN;
    }

    if (xenbus_scanf(XBT_NIL, dev->otherend, "feature-gso-tcpv4", "%d",
             &val) < 0)
        val = 0;
    if (val) {
        be->netif->features |= NETIF_F_TSO;
        be->netif->dev->features |= NETIF_F_TSO;
    }

    if (xenbus_scanf(XBT_NIL, dev->otherend, "feature-gso-tcpv4-prefix", "%d",
             &val) < 0)
        val = 0;
    if (val) {
        be->netif->features |= NETIF_F_TSO;
        be->netif->dev->features |= NETIF_F_TSO;
        be->netif->gso_prefix = 1;
    }

    unregister_csum_offload_watch(be);
    err = register_csum_offload_watch(be);
    if (err)
        return err;

    /* Map the shared frame, irq etc. */
    err = netif_map(be->netif, tx_ring_ref, rx_ring_ref, evtchn);
    if (err) {
        xenbus_dev_fatal(dev, err,
                 "mapping shared-frames %lu/%lu port %u",
                 tx_ring_ref, rx_ring_ref, evtchn);
        return err;
    }
    return 0;
}


/* ** Driver Registration ** */


static const struct xenbus_device_id netback_ids[] = {
    { "vif" },
    { "" }
};


static struct xenbus_driver netback = {
    .name = "vif",
```

```
    .ids = netback_ids,
    .probe = netback_probe,
    .remove = netback_remove,
    .uevent = netback_uevent,
    .otherend_changed = frontend_changed,
};


void netif_xenbus_init(void)
{
    if (xenbus_register_backend(&netback))
        BUG();
}
```

```
/******************************************************************************
 * arch/xen/drivers/netif/backend/interface.c
 *
 * Network-device interface management.
 *
 * Copyright (c) 2004-2005, Keir Fraser
 *
 * 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 "common.h"
#include <linux/ethtool.h>
#include <linux/rtnetlink.h>

/*
 * Module parameter 'queue_length':
 *
 * Enables queuing in the network stack when a client has run out of receive
 * descriptors. Although this feature can improve receive bandwidth by avoiding
 * packet loss, it can also result in packets sitting in the 'tx_queue' for
 * unbounded time. This is bad if those packets hold onto foreign resources.
 * For example, consider a packet that holds onto resources belonging to the
 * guest for which it is queued (e.g., packet received on vif1.0, destined for
 * vif1.1 which is not activated in the guest): in this situation the guest
 * will never be destroyed, unless vif1.1 is taken down. To avoid this, we
 * run a timer (tx_queue_timeout) to drain the queue when the interface is
 * blocked.
 */
static unsigned long netbk_queue_length = 32;
module_param_named(queue_length, netbk_queue_length, ulong, 0644);

static void __netif_up(struct xen_netif *netif)
{
    enable_irq(netif->irq);
    netif_schedule_work(netif);
```

```
}

static void __netif_down(struct xen_netif *netif)
{
    disable_irq(netif->irq);
    netif_deschedule_work(netif);
}

static int net_open(struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);
    if (netback_carrier_ok(netif)) {
        __netif_up(netif);
        netif_start_queue(dev);
    }
    return 0;
}

static int net_close(struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);
    if (netback_carrier_ok(netif))
        __netif_down(netif);
    netif_stop_queue(dev);
    return 0;
}

static int netbk_change_mtu(struct net_device *dev, int mtu)
{
    int max = netbk_can_sg(dev) ? 65535 - ETH_HLEN : ETH_DATA_LEN;

    if (mtu > max)
        return -EINVAL;
    dev->mtu = mtu;
    return 0;
}

static int netbk_set_sg(struct net_device *dev, u32 data)
{
    if (data) {
        struct xen_netif *netif = netdev_priv(dev);

        if (!(netif->features & NETIF_F_SG))
            return -ENOSYS;
    }

    if (dev->mtu > ETH_DATA_LEN)
        dev->mtu = ETH_DATA_LEN;

    return ethtool_op_set_sg(dev, data);
}

static int netbk_set_tso(struct net_device *dev, u32 data)
{
    if (data) {
        struct xen_netif *netif = netdev_priv(dev);
```

```
        if (!(netif->features & NETIF_F_TSO))
            return -ENOSYS;
    }

    return ethtool_op_set_tso(dev, data);
}

static int netbk_set_tx_csum(struct net_device *dev, u32 data)
{
    struct xen_netif *netif = netdev_priv(dev);
    netif_set_tx_csum(netif->be, data);
    return ethtool_op_set_tx_csum(dev, data);
}

static void netbk_get_drvinfo(struct net_device *dev,
                  struct ethtool_drvinfo *info)
{
    strcpy(info->driver, "netbk");
    strcpy(info->bus_info, dev_name(dev->dev.parent));
}

static const struct netif_stat {
    char name[ETH_GSTRING_LEN];
    u16 offset;
} netbk_stats[] = {
    { "copied_skbs", offsetof(struct xen_netif, nr_copied_skbs) },
};

static int netbk_get_stats_count(struct net_device *dev)
{
    return ARRAY_SIZE(netbk_stats);
}

static void netbk_get_ethtool_stats(struct net_device *dev,
                    struct ethtool_stats *stats, u64 * data)
{
    void *netif = netdev_priv(dev);
    int i;

    for (i = 0; i < ARRAY_SIZE(netbk_stats); i++)
        data[i] = *(int *)(netif + netbk_stats[i].offset);
}

static void netbk_get_strings(struct net_device *dev, u32 stringset, u8 * data)
{
    int i;

    switch (stringset) {
    case ETH_SS_STATS:
        for (i = 0; i < ARRAY_SIZE(netbk_stats); i++)
            memcpy(data + i * ETH_GSTRING_LEN,
                   netbk_stats[i].name, ETH_GSTRING_LEN);
        break;
    }
}

static struct ethtool_ops network_ethtool_ops =
```

```
{
    .get_drvinfo = netbk_get_drvinfo,

    .get_tx_csum = ethtool_op_get_tx_csum,
    .set_tx_csum = netbk_set_tx_csum,
    .get_sg = ethtool_op_get_sg,
    .set_sg = netbk_set_sg,
    .get_tso = ethtool_op_get_tso,
    .set_tso = netbk_set_tso,
    .get_link = ethtool_op_get_link,

    .get_stats_count = netbk_get_stats_count,
    .get_ethtool_stats = netbk_get_ethtool_stats,
    .get_strings = netbk_get_strings,
};

struct xen_netif *netif_alloc(struct device *parent, domid_t domid, unsigned int handle)
{
    int err = 0;
    struct net_device *dev;
    struct xen_netif *netif;
    char name[IFNAMSIZ] = {};

    snprintf(name, IFNAMSIZ - 1, "vif%u.%u", domid, handle);
    dev = alloc_netdev(sizeof(struct xen_netif), name, ether_setup);
    if (dev == NULL) {
        DPRINTK("Could not create netif: out of memory\n");
        return ERR_PTR(-ENOMEM);
    }

    SET_NETDEV_DEV(dev, parent);

    netif = netdev_priv(dev);
    memset(netif, 0, sizeof(*netif));
    netif->domid  = domid;
    netif->handle = handle;
    netif->features = NETIF_F_SG;
    atomic_set(&netif->refcnt, 1);
    init_waitqueue_head(&netif->waiting_to_free);
    netif->dev = dev;
    INIT_LIST_HEAD(&netif->list);

    netback_carrier_off(netif);

    netif->credit_bytes = netif->remaining_credit = ~0UL;
    netif->credit_usec  = 0UL;
    init_timer(&netif->credit_timeout);
    /* Initialize 'expires' now: it's used to track the credit window. */
    netif->credit_timeout.expires = jiffies;

    init_timer(&netif->tx_queue_timeout);

    dev->hard_start_xmit = netif_be_start_xmit;
    dev->get_stats       = netif_be_get_stats;
    dev->open            = net_open;
    dev->stop            = net_close;
    dev->change_mtu      = netbk_change_mtu;
```

```
    dev->features        = NETIF_F_IP_CSUM|NETIF_F_SG;

    SET_ETHTOOL_OPS(dev, &network_ethtool_ops);

    dev->tx_queue_len = netbk_queue_length;

    /*
     * Initialise a dummy MAC address. We choose the numerically
     * largest non-broadcast address to prevent the address getting
     * stolen by an Ethernet bridge for STP purposes.
     * (FE:FF:FF:FF:FF:FF)
     */
    memset(dev->dev_addr, 0xFF, ETH_ALEN);
    dev->dev_addr[0] &= ~0x01;

    rtnl_lock();
    err = register_netdevice(dev);
    rtnl_unlock();
    if (err) {
        DPRINTK("Could not register new net device %s: err=%d\n",
            dev->name, err);
        free_netdev(dev);
        return ERR_PTR(err);
    }

    DPRINTK("Successfully created netif\n");
    return netif;
}

static int map_frontend_pages(
    struct xen_netif *netif, grant_ref_t tx_ring_ref, grant_ref_t rx_ring_ref)
{
    struct gnttab_map_grant_ref op;

    gnttab_set_map_op(&op, (unsigned long)netif->tx_comms_area->addr,
              GNTMAP_host_map, tx_ring_ref, netif->domid);

    if (HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref, &op, 1))
        BUG();

    if (op.status) {
        DPRINTK(" Gnttab failure mapping tx_ring_ref!\n");
        return op.status;
    }

    netif->tx_shmem_ref    = tx_ring_ref;
    netif->tx_shmem_handle = op.handle;

    gnttab_set_map_op(&op, (unsigned long)netif->rx_comms_area->addr,
              GNTMAP_host_map, rx_ring_ref, netif->domid);

    if (HYPERVISOR_grant_table_op(GNTTABOP_map_grant_ref, &op, 1))
        BUG();

    if (op.status) {
        struct gnttab_unmap_grant_ref unop;
```

```
        gnttab_set_unmap_op(&unop,
                    (unsigned long)netif->tx_comms_area->addr,
                    GNTMAP_host_map, netif->tx_shmem_handle);
        VOID(HYPERVISOR_grant_table_op(GNTTABOP_unmap_grant_ref,
                            &unop, 1));
        DPRINTK(" Gnttab failure mapping rx_ring_ref!\n");
        return op.status;
    }

    netif->rx_shmem_ref    = rx_ring_ref;
    netif->rx_shmem_handle = op.handle;

    return 0;
}

static void unmap_frontend_pages(struct xen_netif *netif)
{
    struct gnttab_unmap_grant_ref op;

    gnttab_set_unmap_op(&op, (unsigned long)netif->tx_comms_area->addr,
                GNTMAP_host_map, netif->tx_shmem_handle);

    if (HYPERVISOR_grant_table_op(GNTTABOP_unmap_grant_ref, &op, 1))
        BUG();

    gnttab_set_unmap_op(&op, (unsigned long)netif->rx_comms_area->addr,
                GNTMAP_host_map, netif->rx_shmem_handle);

    if (HYPERVISOR_grant_table_op(GNTTABOP_unmap_grant_ref, &op, 1))
        BUG();
}

int netif_map(struct xen_netif *netif, unsigned long tx_ring_ref,
          unsigned long rx_ring_ref, unsigned int evtchn)
{
    int err = -ENOMEM;
    struct xen_netif_tx_sring *txs;
    struct xen_netif_rx_sring *rxs;

    /* Already connected through? */
    if (netif->irq)
        return 0;

    netif->tx_comms_area = alloc_vm_area(PAGE_SIZE);
    if (netif->tx_comms_area == NULL)
        return -ENOMEM;
    netif->rx_comms_area = alloc_vm_area(PAGE_SIZE);
    if (netif->rx_comms_area == NULL)
        goto err_rx;

    err = map_frontend_pages(netif, tx_ring_ref, rx_ring_ref);
    if (err)
        goto err_map;

    err = bind_interdomain_evtchn_to_irqhandler(
        netif->domid, evtchn, netif_be_int, 0,
        netif->dev->name, netif);
```

```
    if (err < 0)
        goto err_hypervisor;
    netif->irq = err;
    disable_irq(netif->irq);

    txs = (struct xen_netif_tx_sring *)netif->tx_comms_area->addr;
    BACK_RING_INIT(&netif->tx, txs, PAGE_SIZE);

    rxs = (struct xen_netif_rx_sring *)
        ((char *)netif->rx_comms_area->addr);
    BACK_RING_INIT(&netif->rx, rxs, PAGE_SIZE);

    netif->rx_req_cons_peek = 0;

    netif_get(netif);

    rtnl_lock();
    netback_carrier_on(netif);
    if (netif_running(netif->dev))
        __netif_up(netif);
    rtnl_unlock();

    return 0;
err_hypervisor:
    unmap_frontend_pages(netif);
err_map:
    free_vm_area(netif->rx_comms_area);
err_rx:
    free_vm_area(netif->tx_comms_area);
    return err;
}

void netif_disconnect(struct xen_netif *netif)
{
    if (netback_carrier_ok(netif)) {
        rtnl_lock();
        netback_carrier_off(netif);
        netif_carrier_off(netif->dev); /* discard queued packets */
        if (netif_running(netif->dev))
            __netif_down(netif);
        rtnl_unlock();
        netif_put(netif);
    }

    atomic_dec(&netif->refcnt);
    wait_event(netif->waiting_to_free, atomic_read(&netif->refcnt) == 0);

    del_timer_sync(&netif->credit_timeout);
    del_timer_sync(&netif->tx_queue_timeout);

    if (netif->irq)
        unbind_from_irqhandler(netif->irq, netif);

    unregister_netdev(netif->dev);

    if (netif->tx.sring) {
        unmap_frontend_pages(netif);
```

```
        free_vm_area(netif->tx_comms_area);
        free_vm_area(netif->rx_comms_area);
    }

    free_netdev(netif->dev);
}
```

```
/******************************************************************************
 * drivers/xen/netback/netback.c
 *
 * Back-end of the driver for virtual network devices. This portion of the
 * driver exports a 'unified' network-device interface that can be accessed
 * by any operating system that implements a compatible front end. A
 * reference front-end implementation can be found in:
 *  drivers/xen/netfront/netfront.c
 *
 * Copyright (c) 2002-2005, K A Fraser
 *
 * 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 "common.h"
#include <xen/balloon.h>
#include <xen/interface/memory.h>
#include <linux/kthread.h>

/*define NETBE_DEBUG_INTERRUPT*/

struct netbk_rx_meta {
    skb_frag_t frag; /* Only if copy == 0 */
    int id;
    int size;
    int gso_size;
    u8 copy:1;
};

struct netbk_tx_pending_inuse {
    struct list_head list;
    unsigned long alloc_time;
};

static void netif_idx_release(u16 pending_idx);
```

```
static void make_tx_response(struct xen_netif *netif,
                             struct xen_netif_tx_request *txp,
                             s8       st);
static struct xen_netif_rx_response *make_rx_response(struct xen_netif *netif,
                                    u16      id,
                                    s8       st,
                                    u16      offset,
                                    u16      size,
                                    u16      flags);

static void net_rx_action(void);
static void net_tx_action(void);
static DECLARE_WAIT_QUEUE_HEAD(netbk_action_wq);

static struct timer_list net_timer;
static struct timer_list netbk_tx_pending_timer;

#define MAX_PENDING_REQS 256

#define MAX_BUFFER_OFFSET PAGE_SIZE

static struct sk_buff_head rx_queue;

static struct page **mmap_pages;
static inline unsigned long idx_to_pfn(unsigned int idx)
{
    return page_to_pfn(mmap_pages[idx]);
}

static inline unsigned long idx_to_kaddr(unsigned int idx)
{
    return (unsigned long)pfn_to_kaddr(idx_to_pfn(idx));
}

/* we use an extra field in struct page to map pages to pending reqs */
#define NETIF_INDEX_MASK 0xABCD0000UL    /* magic sanity check value */
static inline void netif_set_page_index(struct page *pg, unsigned int index)
{
    *(unsigned long *)&pg->mapping = index | NETIF_INDEX_MASK;
}

static inline int netif_page_index(struct page *pg)
{
    unsigned long val = (unsigned long)pg->mapping;
    unsigned long idx = val & ~NETIF_INDEX_MASK;

    if (!PageForeign(pg))
        return -1;

    if ((idx >= MAX_PENDING_REQS) || (mmap_pages[idx] != pg))
        return -1;

    if (unlikely((val & NETIF_INDEX_MASK) != NETIF_INDEX_MASK)) {
        printk(KERN_ERR "apparent netback page %p doesn't have NETIF_INDEX_MASK magic: %
               pg, val, pg->index);
        printk(KERN_ERR "-- page flags are %lx - %s; %s; %s\n",
               pg->flags,
```

```
                PageForeign(pg) ? "FOREIGN" : "not-foreign",
                PageNetback(pg) ? "NETBACK" : "not-netback",
                PageBlkback(pg) ? "BLKBACK" : "not-blkback");
        printk(KERN_ERR "-- page flags are %lx\n", pg->flags);
        printk(KERN_ERR "-- page count is %d\n", page_count(pg));
        BUG();
    }

    return idx;
}

/*
 * This is the amount of packet we copy rather than map, so that the
 * guest can't fiddle with the contents of the headers while we do
 * packet processing on them (netfilter, routing, etc). 72 is enough
 * to cover TCP+IP headers including options.
 */
#define PKT_PROT_LEN 72

static struct pending_tx_info {
    struct xen_netif_tx_request req;
    struct xen_netif *netif;
} pending_tx_info[MAX_PENDING_REQS];
static u16 pending_ring[MAX_PENDING_REQS];
typedef unsigned int pending_ring_idx_t;

static inline pending_ring_idx_t pending_index(unsigned i)
{
    return i & (MAX_PENDING_REQS-1);
}

static pending_ring_idx_t pending_prod, pending_cons;

static inline pending_ring_idx_t nr_pending_reqs(void)
{
    return MAX_PENDING_REQS - pending_prod + pending_cons;
}

/* Freed TX SKBs get batched on this ring before return to pending_ring. */
static u16 dealloc_ring[MAX_PENDING_REQS];
static pending_ring_idx_t dealloc_prod, dealloc_cons;

/* Doubly-linked list of in-use pending entries. */
static struct netbk_tx_pending_inuse pending_inuse[MAX_PENDING_REQS];
static LIST_HEAD(pending_inuse_head);

static struct sk_buff_head tx_queue;

static grant_handle_t grant_tx_handle[MAX_PENDING_REQS];
static gnttab_unmap_grant_ref_t tx_unmap_ops[MAX_PENDING_REQS];
static gnttab_map_grant_ref_t tx_map_ops[MAX_PENDING_REQS];

static LIST_HEAD(net_schedule_list);
static DEFINE_SPINLOCK(net_schedule_list_lock);

#define MAX_MFN_ALLOC 64
static unsigned long mfn_list[MAX_MFN_ALLOC];
```

```
static unsigned int alloc_index = 0;

/* Setting this allows the safe use of this driver without netloop. */
static int MODPARM_copy_skb = 1;
module_param_named(copy_skb, MODPARM_copy_skb, bool, 0);
MODULE_PARM_DESC(copy_skb, "Copy data received from netfront without netloop");

int netbk_copy_skb_mode;

static inline unsigned long alloc_mfn(void)
{
    BUG_ON(alloc_index == 0);
    return mfn_list[--alloc_index];
}

static int check_mfn(int nr)
{
    struct xen_memory_reservation reservation = {
        .extent_order = 0,
        .domid        = DOMID_SELF
    };
    int rc;

    if (likely(alloc_index >= nr))
        return 0;

    set_xen_guest_handle(reservation.extent_start, mfn_list + alloc_index);
    reservation.nr_extents = MAX_MFN_ALLOC - alloc_index;
    rc = HYPERVISOR_memory_op(XENMEM_increase_reservation, &reservation);
    if (likely(rc > 0))
        alloc_index += rc;

    return alloc_index >= nr ? 0 : -ENOMEM;
}

static inline void maybe_schedule_tx_action(void)
{
    smp_mb();
    if ((nr_pending_reqs() < (MAX_PENDING_REQS/2)) &&
        !list_empty(&net_schedule_list))
        wake_up(&netbk_action_wq);
}

static struct sk_buff *netbk_copy_skb(struct sk_buff *skb)
{
    struct skb_shared_info *ninfo;
    struct sk_buff *nskb;
    unsigned long offset;
    int ret;
    int len;
    int headlen;

    BUG_ON(skb_shinfo(skb)->frag_list != NULL);

    nskb = alloc_skb(SKB_MAX_HEAD(0), GFP_ATOMIC | __GFP_NOWARN);
    if (unlikely(!nskb))
        goto err;
```

```
    skb_reserve(nskb, NET_SKB_PAD + NET_IP_ALIGN);
    headlen = skb_end_pointer(nskb) - nskb->data;
    if (headlen > skb_headlen(skb))
        headlen = skb_headlen(skb);
    ret = skb_copy_bits(skb, 0, __skb_put(nskb, headlen), headlen);
    BUG_ON(ret);

    ninfo = skb_shinfo(nskb);
    ninfo->gso_size = skb_shinfo(skb)->gso_size;
    ninfo->gso_type = skb_shinfo(skb)->gso_type;

    offset = headlen;
    len = skb->len - headlen;

    nskb->len = skb->len;
    nskb->data_len = len;
    nskb->truesize += len;

    while (len) {
        struct page *page;
        int copy;
        int zero;

        if (unlikely(ninfo->nr_frags >= MAX_SKB_FRAGS)) {
            dump_stack();
            goto err_free;
        }

        copy = len >= PAGE_SIZE ? PAGE_SIZE : len;
        zero = len >= PAGE_SIZE ? 0 : __GFP_ZERO;

        page = alloc_page(GFP_ATOMIC | __GFP_NOWARN | zero);
        if (unlikely(!page))
            goto err_free;

        ret = skb_copy_bits(skb, offset, page_address(page), copy);
        BUG_ON(ret);

        ninfo->frags[ninfo->nr_frags].page = page;
        ninfo->frags[ninfo->nr_frags].page_offset = 0;
        ninfo->frags[ninfo->nr_frags].size = copy;
        ninfo->nr_frags++;

        offset += copy;
        len -= copy;
    }

#ifdef NET_SKBUFF_DATA_USES_OFFSET
    offset = 0;
#else
    offset = nskb->data - skb->data;
#endif

    nskb->transport_header = skb->transport_header + offset;
    nskb->network_header   = skb->network_header   + offset;
    nskb->mac_header       = skb->mac_header       + offset;
```

```
    return nskb;

 err_free:
    kfree_skb(nskb);
 err:
    return NULL;
}

static inline int netbk_max_required_rx_slots(struct xen_netif *netif)
{
    if (netif->features & (NETIF_F_SG|NETIF_F_TSO))
        return MAX_SKB_FRAGS + 2; /* header + extra_info + frags */
    return 1; /* all in one */
}

static inline int netbk_queue_full(struct xen_netif *netif)
{
    RING_IDX peek   = netif->rx_req_cons_peek;
    RING_IDX needed = netbk_max_required_rx_slots(netif);

    return ((netif->rx.sring->req_prod - peek) < needed) ||
           ((netif->rx.rsp_prod_pvt + NET_RX_RING_SIZE - peek) < needed);
}

static void tx_queue_callback(unsigned long data)
{
    struct xen_netif *netif = (struct xen_netif *)data;
    if (netif_schedulable(netif))
        netif_wake_queue(netif->dev);
}

/* Figure out how many ring slots we're going to need to send @skb to
   the guest. */
static unsigned count_skb_slots(struct sk_buff *skb, struct xen_netif *netif)
{
    unsigned count;
    unsigned copy_off;
    unsigned i;

    copy_off = 0;
    count = 1;

    BUG_ON(offset_in_page(skb->data) + skb_headlen(skb) > MAX_BUFFER_OFFSET);

    copy_off = skb_headlen(skb);

    if (skb_shinfo(skb)->gso_size)
        count++;

    for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
        unsigned long size = skb_shinfo(skb)->frags[i].size;
        unsigned long bytes;
        while (size > 0) {
            BUG_ON(copy_off > MAX_BUFFER_OFFSET);

            /* These checks are the same as in netbk_gop_frag_copy */
```

```
            if (copy_off == MAX_BUFFER_OFFSET
                || ((copy_off + size > MAX_BUFFER_OFFSET) && (size <= MAX_BUFFER_OFFSET)
                count++;
                copy_off = 0;
            }

            bytes = size;
            if (copy_off + bytes > MAX_BUFFER_OFFSET)
                bytes = MAX_BUFFER_OFFSET - copy_off;

            copy_off += bytes;
            size -= bytes;
        }
    }
    return count;
}

int netif_be_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);

    BUG_ON(skb->dev != dev);

    /* Drop the packet if the netif is not up or there is no carrier. */
    if (unlikely(!netif_schedulable(netif)))
        goto drop;

    /* Drop the packet if the target domain has no receive buffers. */
    if (unlikely(netbk_queue_full(netif)))
        goto drop;

    /*
     * Copy the packet here if it's destined for a flipping interface
     * but isn't flippable (e.g. extra references to data).
     * XXX For now we also copy skbuffs whose head crosses a page
     * boundary, because netbk_gop_skb can't handle them.
     */
    if (!netif->copying_receiver ||
        ((skb_headlen(skb) + offset_in_page(skb->data)) >= PAGE_SIZE)) {
        struct sk_buff *nskb = netbk_copy_skb(skb);
        if ( unlikely(nskb == NULL) )
            goto drop;
        /* Copy only the header fields we use in this driver. */
        nskb->dev = skb->dev;
        nskb->ip_summed = skb->ip_summed;
        dev_kfree_skb(skb);
        skb = nskb;
    }

    /* Reserve ring slots for the worst-case number of
     * fragments. */
    netif->rx_req_cons_peek += count_skb_slots(skb, netif);
    netif_get(netif);

    if (netbk_can_queue(dev) && netbk_queue_full(netif)) {
        netif->rx.sring->req_event = netif->rx_req_cons_peek +
            netbk_max_required_rx_slots(netif);
```

```
        mb(); /* request notification /then/ check & stop the queue */
        if (netbk_queue_full(netif)) {
            netif_stop_queue(dev);
            /*
             * Schedule 500ms timeout to restart the queue, thus
             * ensuring that an inactive queue will be drained.
             * Packets will be immediately be dropped until more
             * receive buffers become available (see
             * netbk_queue_full() check above).
             */
            netif->tx_queue_timeout.data = (unsigned long)netif;
            netif->tx_queue_timeout.function = tx_queue_callback;
            mod_timer(&netif->tx_queue_timeout, jiffies + HZ/2);
        }
    }

    skb_queue_tail(&rx_queue, skb);
    wake_up(&netbk_action_wq);

    return 0;

 drop:
    netif->stats.tx_dropped++;
    dev_kfree_skb(skb);
    return 0;
}

struct netrx_pending_operations {
    unsigned trans_prod, trans_cons;
    unsigned mmu_prod, mmu_mcl;
    unsigned mcl_prod, mcl_cons;
    unsigned copy_prod, copy_cons;
    unsigned meta_prod, meta_cons;
    mmu_update_t *mmu;
    gnttab_transfer_t *trans;
    gnttab_copy_t *copy;
    multicall_entry_t *mcl;
    struct netbk_rx_meta *meta;
    int copy_off;
    grant_ref_t copy_gref;
};

/* Set up the grant operations for this fragment.  If it's a flipping
   interface, we also set up the unmap request from here. */

static void netbk_gop_frag_copy(struct xen_netif *netif,
                struct netrx_pending_operations *npo,
                struct page *page, unsigned long size,
                unsigned long offset, int head)
{
    gnttab_copy_t *copy_gop;
    struct netbk_rx_meta *meta;
    int idx = netif_page_index(page);
    unsigned long bytes;

    /* Data must not cross a page boundary. */
    BUG_ON(size + offset > PAGE_SIZE);
```

```
meta = npo->meta + npo->meta_prod - 1;

while (size > 0) {
    BUG_ON(npo->copy_off > MAX_BUFFER_OFFSET);

    /*
     * Move to a new receive buffer if:
     *
     * simple case: we have completely filled the current buffer.
     *
     * complex case: the current frag would overflow
     * the current buffer but only if:
     *     (i)   this frag would fit completely in the next buffer
     * and (ii)  there is already some data in the current buffer
     * and (iii) this is not the head buffer.
     *
     * Where:
     * - (i) stops us splitting a frag into two copies
     *   unless the frag is too large for a single buffer.
     * - (ii) stops us from leaving a buffer pointlessly empty.
     * - (iii) stops us leaving the first buffer
     *   empty. Strictly speaking this is already covered
     *   by (ii) but is explicitly checked because
     *   netfront relies on the first buffer being
     *   non-empty and can crash otherwise.
     *
     * This means we will effectively linearise small
     * frags but do not needlessly split large buffers
     * into multiple copies tend to give large frags their
     * own buffers as before.
     */
    if (npo->copy_off == MAX_BUFFER_OFFSET
        || ((npo->copy_off + size > MAX_BUFFER_OFFSET) && (size <= MAX_BUFFER_OFFSET
        struct xen_netif_rx_request *req;

        BUG_ON(head); /* Netfront requires there to be some data in the head buffer.
        /* Overflowed this request, go to the next one */
        req = RING_GET_REQUEST(&netif->rx, netif->rx.req_cons++);
        meta = npo->meta + npo->meta_prod++;
        meta->gso_size = 0;
        meta->copy = 1;
        meta->size = 0;
        meta->id = req->id;
        npo->copy_off = 0;
        npo->copy_gref = req->gref;
    }

    bytes = size;
    if (npo->copy_off + bytes > MAX_BUFFER_OFFSET)
        bytes = MAX_BUFFER_OFFSET - npo->copy_off;

    copy_gop = npo->copy + npo->copy_prod++;
    copy_gop->flags = GNTCOPY_dest_gref;
    if (idx > -1) {
        struct pending_tx_info *src_pend = &pending_tx_info[idx];
        copy_gop->source.domid = src_pend->netif->domid;
```

```
            copy_gop->source.u.ref = src_pend->req.gref;
            copy_gop->flags |= GNTCOPY_source_gref;
        } else {
            copy_gop->source.domid = DOMID_SELF;
            copy_gop->source.u.gmfn = virt_to_mfn(page_address(page));
        }
        copy_gop->source.offset = offset;
        copy_gop->dest.domid = netif->domid;

        copy_gop->dest.offset = npo->copy_off;
        copy_gop->dest.u.ref = npo->copy_gref;
        copy_gop->len = bytes;

        npo->copy_off += bytes;
        meta->size += bytes;

        offset += bytes;
        size -= bytes;
        head = 0; /* Must be something in this buffer now */
    }
}

static u16 netbk_gop_frag_flip(struct xen_netif *netif, struct netbk_rx_meta *meta,
                   int i, struct netrx_pending_operations *npo,
                   struct page *page, unsigned long size,
                   unsigned long offset)
{
    mmu_update_t *mmu;
    gnttab_transfer_t *gop;
    multicall_entry_t *mcl;
    struct xen_netif_rx_request *req;
    unsigned long old_mfn, new_mfn;

    old_mfn = virt_to_mfn(page_address(page));

    req = RING_GET_REQUEST(&netif->rx, netif->rx.req_cons + i);
    meta->copy = 0;
    if (!xen_feature(XENFEAT_auto_translated_physmap)) {
        new_mfn = alloc_mfn();

        /*
         * Set the new P2M table entry before
         * reassigning the old data page. Heed the
         * comment in pgtable-2level.h:pte_page(). :-)
         */
        set_phys_to_machine(page_to_pfn(page), new_mfn);

        mcl = npo->mcl + npo->mcl_prod++;
        MULTI_update_va_mapping(mcl,
                    (unsigned long)page_address(page),
                    pfn_pte_ma(new_mfn, PAGE_KERNEL),
                    0);

        mmu = npo->mmu + npo->mmu_prod++;
        mmu->ptr = ((maddr_t)new_mfn << PAGE_SHIFT) |
            MMU_MACHPHYS_UPDATE;
        mmu->val = page_to_pfn(page);
```

```
    }

    gop = npo->trans + npo->trans_prod++;
    gop->mfn = old_mfn;
    gop->domid = netif->domid;
    gop->ref = req->gref;

    return req->id;
}

/* Prepare an SKB to be transmitted to the frontend.  This is
   responsible for allocating grant operations, meta structures, etc.
   It returns the number of meta structures consumed.  The number of
   ring slots used is always equal to the number of meta slots used
   plus the number of GSO descriptors used.  Currently, we use either
   zero GSO descriptors (for non-GSO packets) or one descriptor (for
   frontend-side LRO). */
static int netbk_gop_skb(struct sk_buff *skb,
             struct netrx_pending_operations *npo)
{
    struct xen_netif *netif = netdev_priv(skb->dev);
    int nr_frags = skb_shinfo(skb)->nr_frags;
    int i;
    int extra;
    struct xen_netif_rx_request *req;
    struct netbk_rx_meta *head_meta, *meta;
    int old_meta_prod;

    old_meta_prod = npo->meta_prod;

    /* Set up a GSO prefix descriptor, if necessary */
    if (skb_shinfo(skb)->gso_size && netif->gso_prefix) {
        req = RING_GET_REQUEST(&netif->rx, netif->rx.req_cons++);
        meta = npo->meta + npo->meta_prod++;
        meta->gso_size = skb_shinfo(skb)->gso_size;
        meta->size = 0;
        meta->id = req->id;
    }

    if (netif->copying_receiver) {
        req = RING_GET_REQUEST(&netif->rx, netif->rx.req_cons++);
        meta = npo->meta + npo->meta_prod++;

        if (!netif->gso_prefix)
            meta->gso_size = skb_shinfo(skb)->gso_size;
        else
            meta->gso_size = 0;

        meta->copy = 1;
        meta->size = 0;
        meta->id = req->id;
        npo->copy_off = 0;
        npo->copy_gref = req->gref;

        netbk_gop_frag_copy(netif,
                    npo, virt_to_page(skb->data),
                    skb_headlen(skb),
```

```
                    offset_in_page(skb->data), 1);

        if (skb_shinfo(skb)->gso_size && !netif->gso_prefix) {
            /* Leave a gap for the GSO extra descriptor. */
            netif->rx.req_cons++;
        }

        for (i = 0; i < nr_frags; i++) {
            netbk_gop_frag_copy(netif, npo,
                        skb_shinfo(skb)->frags[i].page,
                        skb_shinfo(skb)->frags[i].size,
                        skb_shinfo(skb)->frags[i].page_offset,
                        0);
        }
    } else {
        head_meta = npo->meta + npo->meta_prod++;
        head_meta->frag.page_offset = skb_shinfo(skb)->gso_type;
        head_meta->frag.size = skb_shinfo(skb)->gso_size;
        head_meta->gso_size = skb_shinfo(skb)->gso_size;
        head_meta->size = skb_headlen(skb);
        extra = !!skb_shinfo(skb)->gso_size + 1;

        for (i = 0; i < nr_frags; i++) {
            meta = npo->meta + npo->meta_prod++;
            meta->frag = skb_shinfo(skb)->frags[i];
            meta->id = netbk_gop_frag_flip(netif, meta, i + extra, npo,
                                skb_shinfo(skb)->frags[i].page,
                                skb_shinfo(skb)->frags[i].size,
                                skb_shinfo(skb)->frags[i].page_offset);
        }

        /*
         * This must occur at the end to ensure that we don't
         * trash skb_shinfo until we're done. We know that the
         * head doesn't cross a page boundary because such
         * packets get copied in netif_be_start_xmit.
         */
        head_meta->id = netbk_gop_frag_flip(netif, head_meta, 0, npo,
                                virt_to_page(skb->data),
                                skb_headlen(skb),
                                offset_in_page(skb->data));

        netif->rx.req_cons += nr_frags + extra;
    }
    return npo->meta_prod - old_meta_prod;
}

static inline void netbk_free_pages(int nr_frags, struct netbk_rx_meta *meta)
{
    int i;

    for (i = 0; i < nr_frags; i++)
        put_page(meta[i].frag.page);
}

/* This is a twin to netbk_gop_skb.  Assume that netbk_gop_skb was
   used to set up the operations on the top of
```

```
   netrx_pending_operations, which have since been done.  Check that
   they didn't give any errors and advance over them. */
static int netbk_check_gop(int nr_meta_slots, domid_t domid,
               struct netrx_pending_operations *npo)
{
    multicall_entry_t *mcl;
    gnttab_transfer_t *gop;
    gnttab_copy_t     *copy_op;
    int status = NETIF_RSP_OKAY;
    int i;

    for (i = 0; i < nr_meta_slots; i++) {
        if (npo->meta[npo->meta_cons + i].copy) {
            copy_op = npo->copy + npo->copy_cons++;
            if (copy_op->status != GNTST_okay) {
                DPRINTK("Bad status %d from copy to DOM%d.\n",
                    copy_op->status, domid);
                status = NETIF_RSP_ERROR;
            }
        } else {
            if (!xen_feature(XENFEAT_auto_translated_physmap)) {
                mcl = npo->mcl + npo->mcl_cons++;
                /* The update_va_mapping() must not fail. */
                BUG_ON(mcl->result != 0);
            }

            gop = npo->trans + npo->trans_cons++;
            /* Check the reassignment error code. */
            if (gop->status != 0) {
                DPRINTK("Bad status %d from grant transfer to DOM%u\n",
                    gop->status, domid);
                /*
                 * Page no longer belongs to us unless
                 * GNTST_bad_page, but that should be
                 * a fatal error anyway.
                 */
                BUG_ON(gop->status == GNTST_bad_page);
                status = NETIF_RSP_ERROR;
            }
        }
    }

    return status;
}

static void netbk_add_frag_responses(struct xen_netif *netif, int status,
                     struct netbk_rx_meta *meta,
                     int nr_meta_slots)
{
    int i;
    unsigned long offset;

    for (i = 0; i < nr_meta_slots; i++) {
        int flags;
        if (i == nr_meta_slots - 1)
            flags = 0;
        else
```

```
            flags = NETRXF_more_data;

        if (meta[i].copy)
            offset = 0;
        else
            offset = meta[i].frag.page_offset;
        make_rx_response(netif, meta[i].id, status, offset,
                 meta[i].size, flags);
    }
}

struct skb_cb_overlay {
    int meta_slots_used;
};

static inline int net_rx_action_work_to_do(void)
{
    return !skb_queue_empty(&rx_queue);
}

static void net_rx_action(void)
{
    struct xen_netif *netif = NULL;
    s8 status;
    u16 irq, flags;
    struct xen_netif_rx_response *resp;
    multicall_entry_t *mcl;
    struct sk_buff_head rxq;
    struct sk_buff *skb;
    int notify_nr = 0;
    int ret;
    int nr_frags;
    int count;
    unsigned long offset;
    struct skb_cb_overlay *sco;

    /*
     * Putting hundreds of bytes on the stack is considered rude.
     * Static works because a tasklet can only be on one CPU at any time.
     */
    static multicall_entry_t rx_mcl[NET_RX_RING_SIZE+3];
    static mmu_update_t rx_mmu[NET_RX_RING_SIZE];
    static gnttab_transfer_t grant_trans_op[NET_RX_RING_SIZE];
    /*
     * Each head or fragment can be up to 4096 bytes. Given
     * MAX_BUFFER_OFFSET of 4096 the worst case is that each
     * head/fragment uses 2 copy operation.
     */
    static gnttab_copy_t grant_copy_op[2*NET_RX_RING_SIZE];
    static unsigned char rx_notify[NR_IRQS];
    static u16 notify_list[NET_RX_RING_SIZE];
    static struct netbk_rx_meta meta[NET_RX_RING_SIZE];

    struct netrx_pending_operations npo = {
        mmu: rx_mmu,
        trans: grant_trans_op,
        copy: grant_copy_op,
```

```
    mcl: rx_mcl,
    meta: meta};

skb_queue_head_init(&rxq);

count = 0;

while ((skb = skb_dequeue(&rx_queue)) != NULL) {
    netif = netdev_priv(skb->dev);
    nr_frags = skb_shinfo(skb)->nr_frags;

    if (!netif->copying_receiver &&
        !xen_feature(XENFEAT_auto_translated_physmap) &&
        check_mfn(nr_frags + 1)) {
        /* Memory squeeze? Back off for an arbitrary while. */
        if ( net_ratelimit() )
            WPRINTK("Memory squeeze in netback "
                "driver.\n");
        mod_timer(&net_timer, jiffies + HZ);
        skb_queue_head(&rx_queue, skb);
        break;
    }

    sco = (struct skb_cb_overlay *)skb->cb;
    sco->meta_slots_used = netbk_gop_skb(skb, &npo);

    count += nr_frags + 1;

    __skb_queue_tail(&rxq, skb);

    /* Filled the batch queue? */
    if (count + MAX_SKB_FRAGS >= NET_RX_RING_SIZE)
        break;
}

BUG_ON(npo.meta_prod > ARRAY_SIZE(meta));

npo.mmu_mcl = npo.mcl_prod;
if (npo.mcl_prod) {
    BUG_ON(xen_feature(XENFEAT_auto_translated_physmap));
    BUG_ON(npo.mmu_prod > ARRAY_SIZE(rx_mmu));
    mcl = npo.mcl + npo.mcl_prod++;

    BUG_ON(mcl[-1].op != __HYPERVISOR_update_va_mapping);
    mcl[-1].args[MULTI_UVMFLAGS_INDEX] = UVMF_TLB_FLUSH|UVMF_ALL;

    mcl->op = __HYPERVISOR_mmu_update;
    mcl->args[0] = (unsigned long)rx_mmu;
    mcl->args[1] = npo.mmu_prod;
    mcl->args[2] = 0;
    mcl->args[3] = DOMID_SELF;
}

if (npo.trans_prod) {
    BUG_ON(npo.trans_prod > ARRAY_SIZE(grant_trans_op));
    mcl = npo.mcl + npo.mcl_prod++;
    mcl->op = __HYPERVISOR_grant_table_op;
```

```
        mcl->args[0] = GNTTABOP_transfer;
        mcl->args[1] = (unsigned long)grant_trans_op;
        mcl->args[2] = npo.trans_prod;
    }

    if (npo.copy_prod) {
        BUG_ON(npo.copy_prod > ARRAY_SIZE(grant_copy_op));
        mcl = npo.mcl + npo.mcl_prod++;
        mcl->op = __HYPERVISOR_grant_table_op;
        mcl->args[0] = GNTTABOP_copy;
        mcl->args[1] = (unsigned long)grant_copy_op;
        mcl->args[2] = npo.copy_prod;
    }

    /* Nothing to do? */
    if (!npo.mcl_prod)
        return;

    BUG_ON(npo.mcl_prod > ARRAY_SIZE(rx_mcl));

    ret = HYPERVISOR_multicall(npo.mcl, npo.mcl_prod);
    BUG_ON(ret != 0);
    /* The mmu_machphys_update() must not fail. */
    BUG_ON(npo.mmu_mcl && npo.mcl[npo.mmu_mcl].result != 0);

    while ((skb = __skb_dequeue(&rxq)) != NULL) {
        sco = (struct skb_cb_overlay *)skb->cb;

        netif = netdev_priv(skb->dev);

        if (meta[npo.meta_cons].gso_size && netif->gso_prefix) {
            resp = RING_GET_RESPONSE(&netif->rx, netif->rx.rsp_prod_pvt++);

            resp->flags = NETRXF_gso_prefix | NETRXF_more_data;

            resp->offset = meta[npo.meta_cons].gso_size;
            resp->id = meta[npo.meta_cons].id;
            resp->status = sco->meta_slots_used;

            npo.meta_cons++;
            sco->meta_slots_used--;
        }

        /* We can't rely on skb_release_data to release the
           pages used by fragments for us, since it tries to
           touch the pages in the fraglist.  If we're in
           flipping mode, that doesn't work.  In copying mode,
           we still have access to all of the pages, and so
           it's safe to let release_data deal with it. */
        /* (Freeing the fragments is safe since we copy
           non-linear skbs destined for flipping interfaces) */
        if (!netif->copying_receiver) {
            atomic_set(&(skb_shinfo(skb)->dataref), 1);
            skb_shinfo(skb)->frag_list = NULL;
            skb_shinfo(skb)->nr_frags = 0;
            netbk_free_pages(sco->meta_slots_used - 1,
                             meta + npo.meta_cons + 1);
```

```
}

netif->stats.tx_bytes += skb->len;
netif->stats.tx_packets++;

status = netbk_check_gop(sco->meta_slots_used,
             netif->domid, &npo);

if (sco->meta_slots_used == 1)
    flags = 0;
else
    flags = NETRXF_more_data;

if (skb->ip_summed == CHECKSUM_PARTIAL) /* local packet? */
    flags |= NETRXF_csum_blank | NETRXF_data_validated;
else if (skb->ip_summed == CHECKSUM_UNNECESSARY) /* remote but checksummed? */
    flags |= NETRXF_data_validated;

if (meta[npo.meta_cons].copy)
    offset = 0;
else
    offset = offset_in_page(skb->data);
resp = make_rx_response(netif, meta[npo.meta_cons].id,
            status, offset,
            meta[npo.meta_cons].size,
            flags);

if (meta[npo.meta_cons].gso_size && !netif->gso_prefix) {
    struct xen_netif_extra_info *gso =
        (struct xen_netif_extra_info *)
        RING_GET_RESPONSE(&netif->rx,
                  netif->rx.rsp_prod_pvt++);

    resp->flags |= NETRXF_extra_info;

    gso->u.gso.size = meta[npo.meta_cons].gso_size;
    gso->u.gso.type = XEN_NETIF_GSO_TYPE_TCPV4;
    gso->u.gso.pad = 0;
    gso->u.gso.features = 0;

    gso->type = XEN_NETIF_EXTRA_TYPE_GSO;
    gso->flags = 0;
}

if (sco->meta_slots_used > 1) {
    netbk_add_frag_responses(netif, status,
                 meta + npo.meta_cons + 1,
                 sco->meta_slots_used - 1);
}

RING_PUSH_RESPONSES_AND_CHECK_NOTIFY(&netif->rx, ret);
irq = netif->irq;
if (ret && !rx_notify[irq]) {
    rx_notify[irq] = 1;
    notify_list[notify_nr++] = irq;
}
```

```
        if (netif_queue_stopped(netif->dev) &&
            netif_schedulable(netif) &&
            !netbk_queue_full(netif))
            netif_wake_queue(netif->dev);

        netif_put(netif);
        npo.meta_cons += sco->meta_slots_used;
        dev_kfree_skb(skb);
    }

    while (notify_nr != 0) {
        irq = notify_list[--notify_nr];
        rx_notify[irq] = 0;
        notify_remote_via_irq(irq);
    }

#if 0
    /* More work to do? */
    if (!skb_queue_empty(&rx_queue) && !timer_pending(&net_timer))
        wake_up(&netbk_action_wq);
#endif
}

static void net_alarm(unsigned long unused)
{
    wake_up(&netbk_action_wq);
}

static void netbk_tx_pending_timeout(unsigned long unused)
{
    wake_up(&netbk_action_wq);
}

struct net_device_stats *netif_be_get_stats(struct net_device *dev)
{
    struct xen_netif *netif = netdev_priv(dev);
    return &netif->stats;
}

static int __on_net_schedule_list(struct xen_netif *netif)
{
    return !list_empty(&netif->list);
}

static void remove_from_net_schedule_list(struct xen_netif *netif)
{
    spin_lock_irq(&net_schedule_list_lock);
    if (likely(__on_net_schedule_list(netif))) {
        list_del_init(&netif->list);
        netif_put(netif);
    }
    spin_unlock_irq(&net_schedule_list_lock);
}

static void add_to_net_schedule_list_tail(struct xen_netif *netif)
{
    if (__on_net_schedule_list(netif))
```

```
        return;

    spin_lock_irq(&net_schedule_list_lock);
    if (!__on_net_schedule_list(netif) &&
        likely(netif_schedulable(netif))) {
        list_add_tail(&netif->list, &net_schedule_list);
        netif_get(netif);
    }
    spin_unlock_irq(&net_schedule_list_lock);
}

void netif_schedule_work(struct xen_netif *netif)
{
    int more_to_do;

    RING_FINAL_CHECK_FOR_REQUESTS(&netif->tx, more_to_do);

    if (more_to_do) {
        add_to_net_schedule_list_tail(netif);
        maybe_schedule_tx_action();
    }
}

void netif_deschedule_work(struct xen_netif *netif)
{
    remove_from_net_schedule_list(netif);
}


static void tx_add_credit(struct xen_netif *netif)
{
    unsigned long max_burst, max_credit;

    /*
     * Allow a burst big enough to transmit a jumbo packet of up to 128kB.
     * Otherwise the interface can seize up due to insufficient credit.
     */
    max_burst = RING_GET_REQUEST(&netif->tx, netif->tx.req_cons)->size;
    max_burst = min(max_burst, 131072UL);
    max_burst = max(max_burst, netif->credit_bytes);

    /* Take care that adding a new chunk of credit doesn't wrap to zero. */
    max_credit = netif->remaining_credit + netif->credit_bytes;
    if (max_credit < netif->remaining_credit)
        max_credit = ULONG_MAX; /* wrapped: clamp to ULONG_MAX */

    netif->remaining_credit = min(max_credit, max_burst);
}

static void tx_credit_callback(unsigned long data)
{
    struct xen_netif *netif = (struct xen_netif *)data;
    tx_add_credit(netif);
    netif_schedule_work(netif);
}

static inline int copy_pending_req(pending_ring_idx_t pending_idx)
```

```
{
    int err = gnttab_copy_grant_page(grant_tx_handle[pending_idx],
                    &mmap_pages[pending_idx]);

    if (!err)
        SetPageNetback(mmap_pages[pending_idx]);

    return err;
}

inline static void net_tx_action_dealloc(void)
{
    struct netbk_tx_pending_inuse *inuse, *n;
    gnttab_unmap_grant_ref_t *gop;
    u16 pending_idx;
    pending_ring_idx_t dc, dp;
    struct xen_netif *netif;
    int ret;
    LIST_HEAD(list);

    dc = dealloc_cons;
    gop = tx_unmap_ops;

    /*
     * Free up any grants we have finished using
     */
    do {
        dp = dealloc_prod;

        /* Ensure we see all indices enqueued by netif_idx_release(). */
        smp_rmb();

        while (dc != dp) {
            unsigned long pfn;

            pending_idx = dealloc_ring[pending_index(dc++)];
            list_move_tail(&pending_inuse[pending_idx].list, &list);

            pfn = idx_to_pfn(pending_idx);
            /* Already unmapped? */
            if (!phys_to_machine_mapping_valid(pfn))
                continue;

            gnttab_set_unmap_op(gop, idx_to_kaddr(pending_idx),
                        GNTMAP_host_map,
                        grant_tx_handle[pending_idx]);
            gop++;
        }

        if (netbk_copy_skb_mode != NETBK_DELAYED_COPY_SKB ||
            list_empty(&pending_inuse_head))
            break;

        /* Copy any entries that have been pending for too long. */
        list_for_each_entry_safe(inuse, n, &pending_inuse_head, list) {
            if (time_after(inuse->alloc_time + HZ / 2, jiffies))
                break;
```

```
            pending_idx = inuse - pending_inuse;

            pending_tx_info[pending_idx].netif->nr_copied_skbs++;

            switch (copy_pending_req(pending_idx)) {
            case 0:
                list_move_tail(&inuse->list, &list);
                continue;
            case -EBUSY:
                list_del_init(&inuse->list);
                continue;
            case -ENOENT:
                continue;
            }

            break;
        }
    } while (dp != dealloc_prod);

    dealloc_cons = dc;

    ret = HYPERVISOR_grant_table_op(
        GNTTABOP_unmap_grant_ref, tx_unmap_ops, gop - tx_unmap_ops);
    BUG_ON(ret);

    list_for_each_entry_safe(inuse, n, &list, list) {
        pending_idx = inuse - pending_inuse;

        netif = pending_tx_info[pending_idx].netif;

        make_tx_response(netif, &pending_tx_info[pending_idx].req,
                 NETIF_RSP_OKAY);

        /* Ready for next use. */
        gnttab_reset_grant_page(mmap_pages[pending_idx]);

        pending_ring[pending_index(pending_prod++)] = pending_idx;

        netif_put(netif);

        list_del_init(&inuse->list);
    }
}

static void netbk_tx_err(struct xen_netif *netif, struct xen_netif_tx_request *txp, RING
{
    RING_IDX cons = netif->tx.req_cons;

    do {
        make_tx_response(netif, txp, NETIF_RSP_ERROR);
        if (cons >= end)
            break;
        txp = RING_GET_REQUEST(&netif->tx, cons++);
    } while (1);
    netif->tx.req_cons = cons;
    netif_schedule_work(netif);
```

```
    netif_put(netif);
}

static int netbk_count_requests(struct xen_netif *netif, struct xen_netif_tx_request *fi
                struct xen_netif_tx_request *txp, int work_to_do)
{
    RING_IDX cons = netif->tx.req_cons;
    int frags = 0;

    if (!(first->flags & NETTXF_more_data))
        return 0;

    do {
        if (frags >= work_to_do) {
            DPRINTK("Need more frags\n");
            return -frags;
        }

        if (unlikely(frags >= MAX_SKB_FRAGS)) {
            DPRINTK("Too many frags\n");
            return -frags;
        }

        memcpy(txp, RING_GET_REQUEST(&netif->tx, cons + frags),
               sizeof(*txp));
        if (txp->size > first->size) {
            DPRINTK("Frags galore\n");
            return -frags;
        }

        first->size -= txp->size;
        frags++;

        if (unlikely((txp->offset + txp->size) > PAGE_SIZE)) {
            DPRINTK("txp->offset: %x, size: %u\n",
                txp->offset, txp->size);
            return -frags;
        }
    } while ((txp++)->flags & NETTXF_more_data);

    return frags;
}

static gnttab_map_grant_ref_t *netbk_get_requests(struct xen_netif *netif,
                            struct sk_buff *skb,
                            struct xen_netif_tx_request *txp,
                            gnttab_map_grant_ref_t *mop)
{
    struct skb_shared_info *shinfo = skb_shinfo(skb);
    skb_frag_t *frags = shinfo->frags;
    unsigned long pending_idx = *((u16 *)skb->data);
    int i, start;

    /* Skip first skb fragment if it is on same page as header fragment. */
    start = ((unsigned long)shinfo->frags[0].page == pending_idx);

    for (i = start; i < shinfo->nr_frags; i++, txp++) {
```

```
        pending_idx = pending_ring[pending_index(pending_cons++)];

        gnttab_set_map_op(mop++, idx_to_kaddr(pending_idx),
                  GNTMAP_host_map | GNTMAP_readonly,
                  txp->gref, netif->domid);

        memcpy(&pending_tx_info[pending_idx].req, txp, sizeof(*txp));
        netif_get(netif);
        pending_tx_info[pending_idx].netif = netif;
        frags[i].page = (void *)pending_idx;
    }

    return mop;
}

static int netbk_tx_check_mop(struct sk_buff *skb,
                  gnttab_map_grant_ref_t **mopp)
{
    gnttab_map_grant_ref_t *mop = *mopp;
    int pending_idx = *((u16 *)skb->data);
    struct xen_netif *netif = pending_tx_info[pending_idx].netif;
    struct xen_netif_tx_request *txp;
    struct skb_shared_info *shinfo = skb_shinfo(skb);
    int nr_frags = shinfo->nr_frags;
    int i, err, start;

    /* Check status of header. */
    err = mop->status;
    if (unlikely(err)) {
        txp = &pending_tx_info[pending_idx].req;
        make_tx_response(netif, txp, NETIF_RSP_ERROR);
        pending_ring[pending_index(pending_prod++)] = pending_idx;
        netif_put(netif);
    } else {
        set_phys_to_machine(
            __pa(idx_to_kaddr(pending_idx)) >> PAGE_SHIFT,
            FOREIGN_FRAME(mop->dev_bus_addr >> PAGE_SHIFT));
        grant_tx_handle[pending_idx] = mop->handle;
    }

    /* Skip first skb fragment if it is on same page as header fragment. */
    start = ((unsigned long)shinfo->frags[0].page == pending_idx);

    for (i = start; i < nr_frags; i++) {
        int j, newerr;

        pending_idx = (unsigned long)shinfo->frags[i].page;

        /* Check error status: if okay then remember grant handle. */
        newerr = (++mop)->status;
        if (likely(!newerr)) {
            set_phys_to_machine(
                __pa(idx_to_kaddr(pending_idx))>>PAGE_SHIFT,
                FOREIGN_FRAME(mop->dev_bus_addr>>PAGE_SHIFT));
            grant_tx_handle[pending_idx] = mop->handle;
            /* Had a previous error? Invalidate this fragment. */
            if (unlikely(err))
```

```
                netif_idx_release(pending_idx);
            continue;
        }

        /* Error on this fragment: respond to client with an error. */
        txp = &pending_tx_info[pending_idx].req;
        make_tx_response(netif, txp, NETIF_RSP_ERROR);
        pending_ring[pending_index(pending_prod++)] = pending_idx;
        netif_put(netif);

        /* Not the first error? Preceding frags already invalidated. */
        if (err)
            continue;

        /* First error: invalidate header and preceding fragments. */
        pending_idx = *((u16 *)skb->data);
        netif_idx_release(pending_idx);
        for (j = start; j < i; j++) {
            pending_idx = (unsigned long)shinfo->frags[i].page;
            netif_idx_release(pending_idx);
        }

        /* Remember the error: invalidate all subsequent fragments. */
        err = newerr;
    }

    *mopp = mop + 1;
    return err;
}

static void netbk_fill_frags(struct sk_buff *skb)
{
    struct skb_shared_info *shinfo = skb_shinfo(skb);
    int nr_frags = shinfo->nr_frags;
    int i;

    for (i = 0; i < nr_frags; i++) {
        skb_frag_t *frag = shinfo->frags + i;
        struct xen_netif_tx_request *txp;
        unsigned long pending_idx;

        pending_idx = (unsigned long)frag->page;

        pending_inuse[pending_idx].alloc_time = jiffies;
        list_add_tail(&pending_inuse[pending_idx].list,
                  &pending_inuse_head);

        txp = &pending_tx_info[pending_idx].req;
        frag->page = virt_to_page(idx_to_kaddr(pending_idx));
        frag->size = txp->size;
        frag->page_offset = txp->offset;

        skb->len += txp->size;
        skb->data_len += txp->size;
        skb->truesize += txp->size;
    }
}
```

```
int netbk_get_extras(struct xen_netif *netif, struct xen_netif_extra_info *extras,
             int work_to_do)
{
    struct xen_netif_extra_info extra;
    RING_IDX cons = netif->tx.req_cons;

    do {
        if (unlikely(work_to_do-- <= 0)) {
            DPRINTK("Missing extra info\n");
            return -EBADR;
        }

        memcpy(&extra, RING_GET_REQUEST(&netif->tx, cons),
               sizeof(extra));
        if (unlikely(!extra.type ||
                     extra.type >= XEN_NETIF_EXTRA_TYPE_MAX)) {
            netif->tx.req_cons = ++cons;
            DPRINTK("Invalid extra type: %d\n", extra.type);
            return -EINVAL;
        }

        memcpy(&extras[extra.type - 1], &extra, sizeof(extra));
        netif->tx.req_cons = ++cons;
    } while (extra.flags & XEN_NETIF_EXTRA_FLAG_MORE);

    return work_to_do;
}

static int netbk_set_skb_gso(struct sk_buff *skb, struct xen_netif_extra_info *gso)
{
    if (!gso->u.gso.size) {
        DPRINTK("GSO size must not be zero.\n");
        return -EINVAL;
    }

    /* Currently only TCPv4 S.O. is supported. */
    if (gso->u.gso.type != XEN_NETIF_GSO_TYPE_TCPV4) {
        DPRINTK("Bad GSO type %d.\n", gso->u.gso.type);
        return -EINVAL;
    }

    skb_shinfo(skb)->gso_size = gso->u.gso.size;
    skb_shinfo(skb)->gso_type = SKB_GSO_TCPV4;

    /* Header must be checked, and gso_segs computed. */
    skb_shinfo(skb)->gso_type |= SKB_GSO_DODGY;
    skb_shinfo(skb)->gso_segs = 0;

    return 0;
}

static inline int net_tx_action_work_to_do(void)
{
    if (dealloc_cons != dealloc_prod)
        return 1;
```

```
    if (((nr_pending_reqs() + MAX_SKB_FRAGS) < MAX_PENDING_REQS) &&
        !list_empty(&net_schedule_list))
        return 1;

    return 0;
}

/* Called after netfront has transmitted */
static void net_tx_action(void)
{
    struct list_head *ent;
    struct sk_buff *skb;
    struct xen_netif *netif;
    struct xen_netif_tx_request txreq;
    struct xen_netif_tx_request txfrags[MAX_SKB_FRAGS];
    struct xen_netif_extra_info extras[XEN_NETIF_EXTRA_TYPE_MAX - 1];
    u16 pending_idx;
    RING_IDX i;
    gnttab_map_grant_ref_t *mop;
    unsigned int data_len;
    int ret, work_to_do;

    if (dealloc_cons != dealloc_prod)
        net_tx_action_dealloc();

    mop = tx_map_ops;
    while (((nr_pending_reqs() + MAX_SKB_FRAGS) < MAX_PENDING_REQS) &&
        !list_empty(&net_schedule_list)) {
        /* Get a netif from the list with work to do. */
        ent = net_schedule_list.next;
        netif = list_entry(ent, struct xen_netif, list);
        netif_get(netif);
        remove_from_net_schedule_list(netif);

        RING_FINAL_CHECK_FOR_REQUESTS(&netif->tx, work_to_do);
        if (!work_to_do) {
            netif_put(netif);
            continue;
        }

        i = netif->tx.req_cons;
        rmb(); /* Ensure that we see the request before we copy it. */
        memcpy(&txreq, RING_GET_REQUEST(&netif->tx, i), sizeof(txreq));

        /* Credit-based scheduling. */
        if (txreq.size > netif->remaining_credit) {
            unsigned long now = jiffies;
            unsigned long next_credit =
                netif->credit_timeout.expires +
                msecs_to_jiffies(netif->credit_usec / 1000);

            /* Timer could already be pending in rare cases. */
            if (timer_pending(&netif->credit_timeout)) {
                netif_put(netif);
                continue;
            }
```

```
        /* Passed the point where we can replenish credit? */
        if (time_after_eq(now, next_credit)) {
            netif->credit_timeout.expires = now;
            tx_add_credit(netif);
        }

        /* Still too big to send right now? Set a callback. */
        if (txreq.size > netif->remaining_credit) {
            netif->credit_timeout.data     =
                (unsigned long)netif;
            netif->credit_timeout.function =
                tx_credit_callback;
            mod_timer(&netif->credit_timeout,
                      next_credit);
            netif_put(netif);
            continue;
        }
    }
    netif->remaining_credit -= txreq.size;

    work_to_do--;
    netif->tx.req_cons = ++i;

    memset(extras, 0, sizeof(extras));
    if (txreq.flags & NETTXF_extra_info) {
        work_to_do = netbk_get_extras(netif, extras,
                                      work_to_do);
        i = netif->tx.req_cons;
        if (unlikely(work_to_do < 0)) {
            netbk_tx_err(netif, &txreq, i);
            continue;
        }
    }

    ret = netbk_count_requests(netif, &txreq, txfrags, work_to_do);
    if (unlikely(ret < 0)) {
        netbk_tx_err(netif, &txreq, i - ret);
        continue;
    }
    i += ret;

    if (unlikely(txreq.size < ETH_HLEN)) {
        DPRINTK("Bad packet size: %d\n", txreq.size);
        netbk_tx_err(netif, &txreq, i);
        continue;
    }

    /* No crossing a page as the payload mustn't fragment. */
    if (unlikely((txreq.offset + txreq.size) > PAGE_SIZE)) {
        DPRINTK("txreq.offset: %x, size: %u, end: %lu\n",
            txreq.offset, txreq.size,
            (txreq.offset &~PAGE_MASK) + txreq.size);
        netbk_tx_err(netif, &txreq, i);
        continue;
    }

    pending_idx = pending_ring[pending_index(pending_cons)];
```

```
data_len = (txreq.size > PKT_PROT_LEN &&
        ret < MAX_SKB_FRAGS) ?
    PKT_PROT_LEN : txreq.size;

skb = alloc_skb(data_len + NET_SKB_PAD + NET_IP_ALIGN,
        GFP_ATOMIC | __GFP_NOWARN);
if (unlikely(skb == NULL)) {
    DPRINTK("Can't allocate a skb in start_xmit.\n");
    netbk_tx_err(netif, &txreq, i);
    break;
}

/* Packets passed to netif_rx() must have some headroom. */
skb_reserve(skb, NET_SKB_PAD + NET_IP_ALIGN);

if (extras[XEN_NETIF_EXTRA_TYPE_GSO - 1].type) {
    struct xen_netif_extra_info *gso;
    gso = &extras[XEN_NETIF_EXTRA_TYPE_GSO - 1];

    if (netbk_set_skb_gso(skb, gso)) {
        kfree_skb(skb);
        netbk_tx_err(netif, &txreq, i);
        continue;
    }
}

gnttab_set_map_op(mop, idx_to_kaddr(pending_idx),
          GNTMAP_host_map | GNTMAP_readonly,
          txreq.gref, netif->domid);
mop++;

memcpy(&pending_tx_info[pending_idx].req,
       &txreq, sizeof(txreq));
pending_tx_info[pending_idx].netif = netif;
*((u16 *)skb->data) = pending_idx;

__skb_put(skb, data_len);

skb_shinfo(skb)->nr_frags = ret;
if (data_len < txreq.size) {
    skb_shinfo(skb)->nr_frags++;
    skb_shinfo(skb)->frags[0].page =
        (void *)(unsigned long)pending_idx;
} else {
    /* Discriminate from any valid pending_idx value. */
    skb_shinfo(skb)->frags[0].page = (void *)~0UL;
}

__skb_queue_tail(&tx_queue, skb);

pending_cons++;

mop = netbk_get_requests(netif, skb, txfrags, mop);

netif->tx.req_cons = i;
netif_schedule_work(netif);
```

```
        if ((mop - tx_map_ops) >= ARRAY_SIZE(tx_map_ops))
            break;
    }

    if (mop == tx_map_ops)
        return;

    ret = HYPERVISOR_grant_table_op(
        GNTTABOP_map_grant_ref, tx_map_ops, mop - tx_map_ops);
    BUG_ON(ret);

    mop = tx_map_ops;
    while ((skb = __skb_dequeue(&tx_queue)) != NULL) {
        struct xen_netif_tx_request *txp;

        pending_idx = *((u16 *)skb->data);
        netif       = pending_tx_info[pending_idx].netif;
        txp         = &pending_tx_info[pending_idx].req;

        /* Check the remap error code. */
        if (unlikely(netbk_tx_check_mop(skb, &mop))) {
            DPRINTK("netback grant failed.\n");
            skb_shinfo(skb)->nr_frags = 0;
            kfree_skb(skb);
            continue;
        }

        data_len = skb->len;
        memcpy(skb->data,
               (void *)(idx_to_kaddr(pending_idx)|txp->offset),
               data_len);
        if (data_len < txp->size) {
            /* Append the packet payload as a fragment. */
            txp->offset += data_len;
            txp->size -= data_len;
        } else {
            /* Schedule a response immediately. */
            netif_idx_release(pending_idx);
        }

        if (txp->flags & NETTXF_csum_blank)
            skb->ip_summed = CHECKSUM_PARTIAL;
        else if (txp->flags & NETTXF_data_validated)
            skb->ip_summed = CHECKSUM_UNNECESSARY;

        /*
                 * Workaround Windows frontends which do not set
                 * NETTXF_csum_blank for GSO packets. (CA-31049)
                 */
        if (skb_shinfo(skb)->gso_type)
            skb->ip_summed = CHECKSUM_PARTIAL;

        netbk_fill_frags(skb);

        skb->dev      = netif->dev;
        skb->protocol = eth_type_trans(skb, skb->dev);
```

```
        netif->stats.rx_bytes += skb->len;
        netif->stats.rx_packets++;

        if (skb->ip_summed == CHECKSUM_PARTIAL) {
            if (skb_checksum_setup(skb)) {
                DPRINTK("Can't setup checksum in net_tx_action\n");
                kfree_skb(skb);
                continue;
            }
        } else if (skb_is_gso(skb)) {
            DPRINTK("Dropping GSO but not CHECKSUM_PARTIAL skb\n");
            kfree_skb(skb);
            continue;
        }

        if (unlikely(netbk_copy_skb_mode == NETBK_ALWAYS_COPY_SKB) &&
            unlikely(skb_linearize(skb))) {
            DPRINTK("Can't linearize skb in net_tx_action.\n");
            kfree_skb(skb);
            continue;
        }

        netif_rx_ni(skb);
        netif->dev->last_rx = jiffies;
    }

    if (netbk_copy_skb_mode == NETBK_DELAYED_COPY_SKB &&
        !list_empty(&pending_inuse_head)) {
        struct netbk_tx_pending_inuse *oldest;

        oldest = list_entry(pending_inuse_head.next,
                    struct netbk_tx_pending_inuse, list);
        mod_timer(&netbk_tx_pending_timer, oldest->alloc_time + HZ);
    }
}

static void netif_idx_release(u16 pending_idx)
{
    static DEFINE_SPINLOCK(_lock);
    unsigned long flags;

    spin_lock_irqsave(&_lock, flags);
    dealloc_ring[pending_index(dealloc_prod)] = pending_idx;
    /* Sync with net_tx_action_dealloc: insert idx /then/ incr producer. */
    smp_wmb();
    dealloc_prod++;
    spin_unlock_irqrestore(&_lock, flags);

    wake_up(&netbk_action_wq);
}

static void netif_page_release(struct page *page, unsigned int order)
{
    int idx = netif_page_index(page);
    BUG_ON(order);
    BUG_ON(idx < 0);
```

```
    netif_idx_release(idx);
}

irqreturn_t netif_be_int(int irq, void *dev_id)
{
    struct xen_netif *netif = dev_id;

    add_to_net_schedule_list_tail(netif);
    maybe_schedule_tx_action();

    if (netif_schedulable(netif) && !netbk_queue_full(netif))
        netif_wake_queue(netif->dev);

    return IRQ_HANDLED;
}

static void make_tx_response(struct xen_netif *netif,
                 struct xen_netif_tx_request *txp,
                 s8       st)
{
    RING_IDX i = netif->tx.rsp_prod_pvt;
    struct xen_netif_tx_response *resp;
    int notify;

    resp = RING_GET_RESPONSE(&netif->tx, i);
    resp->id     = txp->id;
    resp->status = st;

    if (txp->flags & NETTXF_extra_info)
        RING_GET_RESPONSE(&netif->tx, ++i)->status = NETIF_RSP_NULL;

    netif->tx.rsp_prod_pvt = ++i;
    RING_PUSH_RESPONSES_AND_CHECK_NOTIFY(&netif->tx, notify);
    if (notify)
        notify_remote_via_irq(netif->irq);
}

static struct xen_netif_rx_response *make_rx_response(struct xen_netif *netif,
                          u16      id,
                          s8       st,
                          u16      offset,
                          u16      size,
                          u16      flags)
{
    RING_IDX i = netif->rx.rsp_prod_pvt;
    struct xen_netif_rx_response *resp;

    resp = RING_GET_RESPONSE(&netif->rx, i);
    resp->offset     = offset;
    resp->flags      = flags;
    resp->id         = id;
    resp->status     = (s16)size;
    if (st < 0)
        resp->status = (s16)st;

    netif->rx.rsp_prod_pvt = ++i;
```

```
    return resp;
}

static int netbk_action_thread(void *unused)
{
    while (1) {
        wait_event_interruptible(netbk_action_wq,
            net_rx_action_work_to_do() || net_tx_action_work_to_do());
        cond_resched();

        if (net_rx_action_work_to_do())
            net_rx_action();

        if (net_tx_action_work_to_do())
            net_tx_action();
    }

    return 0;
}


#ifdef NETBE_DEBUG_INTERRUPT
static irqreturn_t netif_be_dbg(int irq, void *dev_id)
{
    struct list_head *ent;
    struct xen_netif *netif;
    int i = 0;

    printk(KERN_ALERT "netif_schedule_list:\n");
    spin_lock_irq(&net_schedule_list_lock);

    list_for_each (ent, &net_schedule_list) {
        netif = list_entry(ent, struct xen_netif, list);
        printk(KERN_ALERT " %d: private(rx_req_cons=%08x "
               "rx_resp_prod=%08x\n",
               i, netif->rx.req_cons, netif->rx.rsp_prod_pvt);
        printk(KERN_ALERT "   tx_req_cons=%08x tx_resp_prod=%08x)\n",
               netif->tx.req_cons, netif->tx.rsp_prod_pvt);
        printk(KERN_ALERT "   shared(rx_req_prod=%08x "
               "rx_resp_prod=%08x\n",
               netif->rx.sring->req_prod, netif->rx.sring->rsp_prod);
        printk(KERN_ALERT "   rx_event=%08x tx_req_prod=%08x\n",
               netif->rx.sring->rsp_event, netif->tx.sring->req_prod);
        printk(KERN_ALERT "   tx_resp_prod=%08x, tx_event=%08x)\n",
               netif->tx.sring->rsp_prod, netif->tx.sring->rsp_event);
        i++;
    }

    spin_unlock_irq(&net_schedule_list_lock);
    printk(KERN_ALERT " ** End of netif_schedule_list **\n");

    return IRQ_HANDLED;
}

static struct irqaction netif_be_dbg_action = {
    .handler = netif_be_dbg,
    .flags   = IRQF_SHARED,
```

```
    .name     = "net-be-dbg"
};
#endif

static int __init netback_init(void)
{
    int i;
    struct page *page;
    struct task_struct *task;

    if (!is_running_on_xen())
        return -ENODEV;

    /* We can increase reservation by this much in net_rx_action(). */
    balloon_update_driver_allowance(NET_RX_RING_SIZE);

    skb_queue_head_init(&rx_queue);
    skb_queue_head_init(&tx_queue);

    init_timer(&net_timer);
    net_timer.data = 0;
    net_timer.function = net_alarm;

    init_timer(&netbk_tx_pending_timer);
    netbk_tx_pending_timer.data = 0;
    netbk_tx_pending_timer.function = netbk_tx_pending_timeout;

    mmap_pages = alloc_empty_pages_and_pagevec(MAX_PENDING_REQS);
    if (mmap_pages == NULL) {
        printk("%s: out of memory\n", __FUNCTION__);
        return -ENOMEM;
    }

    for (i = 0; i < MAX_PENDING_REQS; i++) {
        page = mmap_pages[i];
        SetPageForeign(page, netif_page_release);
        SetPageNetback(page);
        netif_set_page_index(page, i);
        INIT_LIST_HEAD(&pending_inuse[i].list);
    }

    pending_cons = 0;
    pending_prod = MAX_PENDING_REQS;
    for (i = 0; i < MAX_PENDING_REQS; i++)
        pending_ring[i] = i;

    netbk_copy_skb_mode = NETBK_DONT_COPY_SKB;
    if (MODPARM_copy_skb) {
        if (HYPERVISOR_grant_table_op(GNTTABOP_unmap_and_replace,
                                      NULL, 0))
            netbk_copy_skb_mode = NETBK_ALWAYS_COPY_SKB;
        else
            netbk_copy_skb_mode = NETBK_DELAYED_COPY_SKB;
    }

    netif_accel_init();
```

```
    netif_xenbus_init();

    task = kthread_run(netbk_action_thread, NULL, "netback");
    if (IS_ERR(task))
        return PTR_ERR(task);

#ifdef NETBE_DEBUG_INTERRUPT
    (void)bind_virq_to_irqaction(VIRQ_DEBUG,
                    0,
                    &netif_be_dbg_action);
#endif

    return 0;
}

module_init(netback_init);

MODULE_LICENSE("Dual BSD/GPL");
```