```
**********************************************************
  119906 Thu Dec 22 14:34:36 2011
new/usr/src/uts/common/io/mac/mac_sched.c
1925 stack overflow from mac code
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 * Copyright 2011 Joyent, Inc.  All rights reserved.
 */

#include <sys/types.h>
#include <sys/callb.h>
#include <sys/sdt.h>
#include <sys/strsubr.h>
#include <sys/strsun.h>
#include <sys/vlan.h>
#include <sys/stack.h>
#include <sys/archsystm.h>
#include <inet/ipsec_impl.h>
#include <inet/ip_impl.h>
#include <inet/sadb.h>
#include <inet/ipsecesp.h>
#include <inet/ipsecah.h>
#include <inet/ip6.h>

#include <sys/mac_impl.h>
#include <sys/mac_client_impl.h>
#include <sys/mac_client_priv.h>
#include <sys/mac_soft_ring.h>
#include <sys/mac_flow_impl.h>

static mac_tx_cookie_t mac_tx_single_ring_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_serializer_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_fanout_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_bw_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);
static mac_tx_cookie_t mac_tx_aggr_mode(mac_soft_ring_set_t *, mblk_t *,
    uintptr_t, uint16_t, mblk_t **);

typedef struct mac_tx_mode_s {
```



```
        mac_tx_srs_mode_t       mac_tx_mode;
        mac_tx_func_t           mac_tx_func;
} mac_tx_mode_t;
______unchanged_portion_omitted_

/*
 * MAC_RX_SRS_TOODEEP
 *
 * Macro called as part of receive-side processing to determine if handling
 * can occur in situ (in the interrupt thread) or if it should be left to a
 * worker thread.  Note that the constant used to make this determination is
 * not entirely made-up, and is a result of some emprical validation. That
 * said, the constant is left as a static variable to allow it to be
 * dynamically tuned in the field if and as needed.
 */
static uintptr_t mac_rx_srs_stack_needed = 10240;
static uint_t mac_rx_srs_stack_toodeep;

#ifndef STACK_GROWTH_DOWN
#error Downward stack growth assumed.
#endif

#define MAC_RX_SRS_TOODEEP() (STACK_BIAS + (uintptr_t)getfp() - \
        (uintptr_t)curthread->t_stkbase < mac_rx_srs_stack_needed && \
        ++mac_rx_srs_stack_toodeep)


/*
 * Drop the rx packet and advance to the next one in the chain.
 */
static void
mac_rx_drop_pkt(mac_soft_ring_set_t *srs, mblk_t *mp)
{
        mac_srs_rx_t    *srs_rx = &srs->srs_rx;

        ASSERT(mp->b_next == NULL);
        mutex_enter(&srs->srs_lock);
        MAC_UPDATE_SRS_COUNT_LOCKED(srs, 1);
        MAC_UPDATE_SRS_SIZE_LOCKED(srs, msgdsize(mp));
        mutex_exit(&srs->srs_lock);

        srs_rx->sr_stat.mrs_sdrops++;
        freemsg(mp);
}
______unchanged_portion_omitted_

/*
 * mac_rx_srs_process
 *
 * Receive side routine called from the interrupt path.
 *
 * loopback is set to force a context switch on the loopback
 * path between MAC clients.
 */
/* ARGSUSED */
void
mac_rx_srs_process(void *arg, mac_resource_handle_t srs, mblk_t *mp_chain,
    boolean_t loopback)
{
        mac_soft_ring_set_t     *mac_srs = (mac_soft_ring_set_t *)srs;
        mblk_t                  *mp, *tail, *head;
        int                     count = 0;
        int                     count1;
        size_t                  sz = 0;
        size_t                  chain_sz, sz1;
        mac_bw_ctl_t            *mac_bw;
```

```
2333         mac_srs_rx_t            *srs_rx = &mac_srs->srs_rx;

2335         /*
2336          * Set the tail, count and sz. We set the sz irrespective
2337          * of whether we are doing B/W control or not for the
2338          * purpose of updating the stats.
2339          */
2340         mp = tail = mp_chain;
2341         while (mp != NULL) {
2342                 tail = mp;
2343                 count++;
2344                 sz += msgdsize(mp);
2345                 mp = mp->b_next;
2346         }

2348         mutex_enter(&mac_srs->srs_lock);

2350         if (loopback) {
2351                 SRS_RX_STAT_UPDATE(mac_srs, lclbytes, sz);
2352                 SRS_RX_STAT_UPDATE(mac_srs, lclcnt, count);

2354         } else {
2355                 SRS_RX_STAT_UPDATE(mac_srs, intrbytes, sz);
2356                 SRS_RX_STAT_UPDATE(mac_srs, intrcnt, count);
2357         }

2359         /*
2360          * If the SRS in already being processed; has been blanked;
2361          * can be processed by worker thread only; or the B/W limit
2362          * has been reached, then queue the chain and check if
2363          * worker thread needs to be awakend.
2364          */
2365         if (mac_srs->srs_type & SRST_BW_CONTROL) {
2366                 mac_bw = mac_srs->srs_bw;
2367                 ASSERT(mac_bw != NULL);
2368                 mutex_enter(&mac_bw->mac_bw_lock);
2369                 mac_bw->mac_bw_intr += sz;
2370                 if (mac_bw->mac_bw_limit == 0) {
2371                         /* zero bandwidth: drop all */
2372                         srs_rx->sr_stat.mrs_sdrops += count;
2373                         mac_bw->mac_bw_drop_bytes += sz;
2374                         mutex_exit(&mac_bw->mac_bw_lock);
2375                         mutex_exit(&mac_srs->srs_lock);
2376                         mac_pkt_drop(NULL, NULL, mp_chain, B_FALSE);
2377                         return;
2378                 } else {
2379                         if ((mac_bw->mac_bw_sz + sz) <=
2380                             mac_bw->mac_bw_drop_threshold) {
2381                                 mutex_exit(&mac_bw->mac_bw_lock);
2382                                 MAC_RX_SRS_ENQUEUE_CHAIN(mac_srs, mp_chain,
2383                                     tail, count, sz);
2384                         } else {
2385                                 mp = mp_chain;
2386                                 chain_sz = 0;
2387                                 count1 = 0;
2388                                 tail = NULL;
2389                                 head = NULL;
2390                                 while (mp != NULL) {
2391                                         sz1 = msgdsize(mp);
2392                                         if (mac_bw->mac_bw_sz + chain_sz + sz1 >
2393                                             mac_bw->mac_bw_drop_threshold)
2394                                                 break;
2395                                         chain_sz += sz1;
2396                                         count1++;
2397                                         tail = mp;
2398                                         mp = mp->b_next;
```



```
2399                                 }
2400                                 mutex_exit(&mac_bw->mac_bw_lock);
2401                                 if (tail != NULL) {
2402                                         head = tail->b_next;
2403                                         tail->b_next = NULL;
2404                                         MAC_RX_SRS_ENQUEUE_CHAIN(mac_srs,
2405                                             mp_chain, tail, count1, chain_sz);
2406                                         sz -= chain_sz;
2407                                         count -= count1;
2408                                 } else {
2409                                         /* Can't pick up any */
2410                                         head = mp_chain;
2411                                 }
2412                                 if (head != NULL) {
2413                                         /* Drop any packet over the threshold */
2414                                         srs_rx->sr_stat.mrs_sdrops += count;
2415                                         mutex_enter(&mac_bw->mac_bw_lock);
2416                                         mac_bw->mac_bw_drop_bytes += sz;
2417                                         mutex_exit(&mac_bw->mac_bw_lock);
2418                                         freemsgchain(head);
2419                                 }
2420                         }
2421                         MAC_SRS_WORKER_WAKEUP(mac_srs);
2422                         mutex_exit(&mac_srs->srs_lock);
2423                         return;
2424                 }
2425         }

2427         /*
2428          * If the total number of packets queued in the SRS and
2429          * its associated soft rings exceeds the max allowed,
2430          * then drop the chain. If we are polling capable, this
2431          * shouldn't be happening.
2432          */
2433         if (!(mac_srs->srs_type & SRST_BW_CONTROL) &&
2434             (srs_rx->sr_poll_pkt_cnt > srs_rx->sr_hiwat)) {
2435                 mac_bw = mac_srs->srs_bw;
2436                 srs_rx->sr_stat.mrs_sdrops += count;
2437                 mutex_enter(&mac_bw->mac_bw_lock);
2438                 mac_bw->mac_bw_drop_bytes += sz;
2439                 mutex_exit(&mac_bw->mac_bw_lock);
2440                 freemsgchain(mp_chain);
2441                 mutex_exit(&mac_srs->srs_lock);
2442                 return;
2443         }

2445         MAC_RX_SRS_ENQUEUE_CHAIN(mac_srs, mp_chain, tail, count, sz);

2447         if (!(mac_srs->srs_state & SRS_PROC)) {
2448                 /*
2449                  * If we are coming via loopback, if we are not optimizing for
2450                  * latency, or if our stack is running deep, we should signal
2451                  * the worker thread.
2422                  * If we are coming via loopback or if we are not
2423                  * optimizing for latency, we should signal the
2424                  * worker thread.
2452                  */
2453                 if (loopback || !(mac_srs->srs_state & SRS_LATENCY_OPT) ||
2454                     MAC_RX_SRS_TOODEEP()) {
2426                 if (loopback || !(mac_srs->srs_state & SRS_LATENCY_OPT)) {
2455                         /*
2456                          * For loopback, We need to let the worker take
2457                          * over as we don't want to continue in the same
2458                          * thread even if we can. This could lead to stack
2459                          * overflows and may also end up using
2460                          * resources (cpu) incorrectly.
```

```
2461                            */
2462                           cv_signal(&mac_srs->srs_async);
2463                   } else {
2464                           /*
2465                            * Seems like no one is processing the SRS and
2466                            * there is no backlog. We also inline process
2467                            * our packet if its a single packet in non
2468                            * latency optimized case (in latency optimized
2469                            * case, we inline process chains of any size).
2470                            */
2471                           mac_srs->srs_drain_func(mac_srs, SRS_PROC_FAST);
2472                   }
2473           }
2474           mutex_exit(&mac_srs->srs_lock);
2475 }
```

unchanged_portion_omitted_