Linux Cross Reference

Free Electrons

Embedded Linux Experts

Linux/include/net/dst.h

```
 1 /*
 2  * net/dst.h    Protocol independent destination cache definitions.
 3  *
 4  * Authors:     Alexey Kuznetsov, <kuznet@ms2.inr.ac.ru>
 5  *
 6  */
 7
 8 #ifndef _NET_DST_H
 9 #define _NET_DST_H
10
11 #include <net/dst_ops.h>
12 #include <linux/netdevice.h>
13 #include <linux/rtnetlink.h>
14 #include <linux/rcupdate.h>
15 #include <linux/bug.h>
16 #include <linux/jiffies.h>
17 #include <net/neighbour.h>
18 #include <asm/processor.h>
19
20 #define DST_GC_MIN      (HZ/10)
21 #define DST_GC_INC      (HZ/2)
22 #define DST_GC_MAX      (120*HZ)
23
24 /* Each dst_entry has reference count and sits in some parent list(s).
25  * When it is removed from parent list, it is "freed" (dst_free).
26  * After this it enters dead state (dst->obsolete > 0) and if its refcnt
27  * is zero, it can be destroyed immediately, otherwise it is added
28  * to gc list and garbage collector periodically checks the refcnt.
29  */
30
31 struct sk_buff;
32
33 struct dst_entry {
34         struct rcu_head         rcu_head;
35         struct dst_entry        *child;
36         struct net_device       *dev;
37         struct  dst_ops         *ops;
38         unsigned long           _metrics;
39         unsigned long           expires;
40         struct dst_entry        *path;
41         struct dst_entry        *from;
42 #ifdef CONFIG_XFRM
43         struct xfrm_state       *xfrm;
44 #else
45         void                    *__pad1;
46 #endif
47         int                     (*input)(struct sk_buff *);
48         int                     (*output)(struct sock *sk, struct sk_buff *skb);
49
```

```
 50         unsigned short          flags;
 51 #define DST_HOST                0x0001
 52 #define DST_NOXFRM              0x0002
 53 #define DST_NOPOLICY            0x0004
 54 #define DST_NOHASH              0x0008
 55 #define DST_NOCACHE             0x0010
 56 #define DST_NOCOUNT             0x0020
 57 #define DST_FAKE_RTABLE         0x0040
 58 #define DST_XFRM_TUNNEL         0x0080
 59 #define DST_XFRM_QUEUE          0x0100
 60
 61         unsigned short          pending_confirm;
 62
 63         short                   error;
 64
 65         /* A non-zero value of dst->obsolete forces by-hand validation
 66          * of the route entry.  Positive values are set by the generic
 67          * dst layer to indicate that the entry has been forcefully
 68          * destroyed.
 69          *
 70          * Negative values are used by the implementation layer code to
 71          * force invocation of the dst_ops->check() method.
 72          */
 73         short                   obsolete;
 74 #define DST_OBSOLETE_NONE       0
 75 #define DST_OBSOLETE_DEAD       2
 76 #define DST_OBSOLETE_FORCE_CHK  -1
 77 #define DST_OBSOLETE_KILL       -2
 78         unsigned short          header_len;     /* more space at head required */
 79         unsigned short          trailer_len;    /* space to reserve at tail */
 80 #ifdef CONFIG_IP_ROUTE_CLASSID
 81         __u32                   tclassid;
 82 #else
 83         __u32                   __pad2;
 84 #endif
 85
 86         /*
 87          * Align __refcnt to a 64 bytes alignment
 88          * (L1_CACHE_SIZE would be too much)
 89          */
 90 #ifdef CONFIG_64BIT
 91         long                    __pad_to_align_refcnt[2];
 92 #endif
 93         /*
 94          * __refcnt wants to be on a different cache line from
 95          * input/output/ops or performance tanks badly
 96          */
 97         atomic_t                __refcnt;       /* client references    */
 98         int                     __use;
 99         unsigned long           lastuse;
100         union {
101                 struct dst_entry        *next;
102                 struct rtable __rcu     *rt_next;
103                 struct rt6_info         *rt6_next;
104                 struct dn_route __rcu   *dn_next;
105         };
106 };
107
108 u32 *dst_cow_metrics_generic(struct dst_entry *dst, unsigned long old);
109 extern const u32 dst_default_metrics[];
110
111 #define DST_METRICS_READ_ONLY           0x1UL
112 #define DST_METRICS_FLAGS               0x3UL
113 #define __DST_METRICS_PTR(Y)    \
114         ((u32 *)((Y) & ~DST_METRICS_FLAGS))
115 #define DST_METRICS_PTR(X)      __DST_METRICS_PTR((X)->_metrics)
116
117 static inline bool dst_metrics_read_only(const struct dst_entry *dst)
118 {
119         return dst->_metrics & DST_METRICS_READ_ONLY;
```

```
120 }
121
122 void __dst_destroy_metrics_generic(struct dst_entry *dst, unsigned long old);
123
124 static inline void dst_destroy_metrics_generic(struct dst_entry *dst)
125 {
126         unsigned long val = dst->_metrics;
127         if (!(val & DST_METRICS_READ_ONLY))
128                 __dst_destroy_metrics_generic(dst, val);
129 }
130
131 static inline u32 *dst_metrics_write_ptr(struct dst_entry *dst)
132 {
133         unsigned long p = dst->_metrics;
134
135         BUG_ON(!p);
136
137         if (p & DST_METRICS_READ_ONLY)
138                 return dst->ops->cow_metrics(dst, p);
139         return __DST_METRICS_PTR(p);
140 }
141
142 /* This may only be invoked before the entry has reached global
143  * visibility.
144  */
145 static inline void dst_init_metrics(struct dst_entry *dst,
146                                     const u32 *src_metrics,
147                                     bool read_only)
148 {
149         dst->_metrics = ((unsigned long) src_metrics) |
150                 (read_only ? DST_METRICS_READ_ONLY : 0);
151 }
152
153 static inline void dst_copy_metrics(struct dst_entry *dest, const struct dst_entry *src)
154 {
155         u32 *dst_metrics = dst_metrics_write_ptr(dest);
156
157         if (dst_metrics) {
158                 u32 *src_metrics = DST_METRICS_PTR(src);
159
160                 memcpy(dst_metrics, src_metrics, RTAX_MAX * sizeof(u32));
161         }
162 }
163
164 static inline u32 *dst_metrics_ptr(struct dst_entry *dst)
165 {
166         return DST_METRICS_PTR(dst);
167 }
168
169 static inline u32
170 dst_metric_raw(const struct dst_entry *dst, const int metric)
171 {
172         u32 *p = DST_METRICS_PTR(dst);
173
174         return p[metric-1];
175 }
176
177 static inline u32
178 dst_metric(const struct dst_entry *dst, const int metric)
179 {
180         WARN_ON_ONCE(metric == RTAX_HOPLIMIT ||
181                      metric == RTAX_ADVMSS ||
182                      metric == RTAX_MTU);
183         return dst_metric_raw(dst, metric);
184 }
185
186 static inline u32
187 dst_metric_advmss(const struct dst_entry *dst)
188 {
189         u32 advmss = dst_metric_raw(dst, RTAX_ADVMSS);
```

```
190
191         if (!advmss)
192                 advmss = dst->ops->default_advmss(dst);
193
194         return advmss;
195 }
196
197 static inline void dst_metric_set(struct dst_entry *dst, int metric, u32 val)
198 {
199         u32 *p = dst_metrics_write_ptr(dst);
200
201         if (p)
202                 p[metric-1] = val;
203 }
204
205 static inline u32
206 dst_feature(const struct dst_entry *dst, u32 feature)
207 {
208         return dst_metric(dst, RTAX_FEATURES) & feature;
209 }
210
211 static inline u32 dst_mtu(const struct dst_entry *dst)
212 {
213         return dst->ops->mtu(dst);
214 }
215
216 /* RTT metrics are stored in milliseconds for user ABI, but used as jiffies */
217 static inline unsigned long dst_metric_rtt(const struct dst_entry *dst, int metric)
218 {
219         return msecs_to_jiffies(dst_metric(dst, metric));
220 }
221
222 static inline u32
223 dst_allfrag(const struct dst_entry *dst)
224 {
225         int ret = dst_feature(dst,  RTAX_FEATURE_ALLFRAG);
226         return ret;
227 }
228
229 static inline int
230 dst_metric_locked(const struct dst_entry *dst, int metric)
231 {
232         return dst_metric(dst, RTAX_LOCK) & (1<<metric);
233 }
234
235 static inline void dst_hold(struct dst_entry *dst)
236 {
237         /*
238          * If your kernel compilation stops here, please check
239          * __pad_to_align_refcnt declaration in struct dst_entry
240          */
241         BUILD_BUG_ON(offsetof(struct dst_entry, __refcnt) & 63);
242         atomic_inc(&dst->__refcnt);
243 }
244
245 static inline void dst_use(struct dst_entry *dst, unsigned long time)
246 {
247         dst_hold(dst);
248         dst->__use++;
249         dst->lastuse = time;
250 }
251
252 static inline void dst_use_noref(struct dst_entry *dst, unsigned long time)
253 {
254         dst->__use++;
255         dst->lastuse = time;
256 }
257
258 static inline struct dst_entry *dst_clone(struct dst_entry *dst)
259 {
```

```
260         if (dst)
261                 atomic_inc(&dst->__refcnt);
262         return dst;
263 }
264
265 void dst_release(struct dst_entry *dst);
266
267 static inline void refdst_drop(unsigned long refdst)
268 {
269         if (!(refdst & SKB_DST_NOREF))
270                 dst_release((struct dst_entry *)(refdst & SKB_DST_PTRMASK));
271 }
272
273 /**
274  * skb_dst_drop - drops skb dst
275  * @skb: buffer
276  *
277  * Drops dst reference count if a reference was taken.
278  */
279 static inline void skb_dst_drop(struct sk_buff *skb)
280 {
281         if (skb->_skb_refdst) {
282                 refdst_drop(skb->_skb_refdst);
283                 skb->_skb_refdst = 0UL;
284         }
285 }
286
287 static inline void skb_dst_copy(struct sk_buff *nskb, const struct sk_buff *oskb)
288 {
289         nskb->_skb_refdst = oskb->_skb_refdst;
290         if (!(nskb->_skb_refdst & SKB_DST_NOREF))
291                 dst_clone(skb_dst(nskb));
292 }
293
294 /**
295  * skb_dst_force - makes sure skb dst is refcounted
296  * @skb: buffer
297  *
298  * If dst is not yet refcounted, let's do it
299  */
300 static inline void skb_dst_force(struct sk_buff *skb)
301 {
302         if (skb_dst_is_noref(skb)) {
303                 WARN_ON(!rcu_read_lock_held());
304                 skb->_skb_refdst &= ~SKB_DST_NOREF;
305                 dst_clone(skb_dst(skb));
306         }
307 }
308
309
310 /**
311  *      __skb_tunnel_rx - prepare skb for rx reinsert
312  *      @skb: buffer
313  *      @dev: tunnel device
314  *      @net: netns for packet i/o
315  *
316  *      After decapsulation, packet is going to re-enter (netif_rx()) our stack,
317  *      so make some cleanups. (no accounting done)
318  */
319 static inline void __skb_tunnel_rx(struct sk_buff *skb, struct net_device *dev,
320                                    struct net *net)
321 {
322         skb->dev = dev;
323
324         /*
325          * Clear hash so that we can recalulate the hash for the
326          * encapsulated packet, unless we have already determine the hash
327          * over the L4 4-tuple.
328          */
329         skb_clear_hash_if_not_l4(skb);
```

```
330         skb_set_queue_mapping(skb, 0);
331         skb_scrub_packet(skb, !net_eq(net, dev_net(dev)));
332 }
333
334 /**
335  *      skb_tunnel_rx - prepare skb for rx reinsert
336  *      @skb: buffer
337  *      @dev: tunnel device
338  *
339  *      After decapsulation, packet is going to re-enter (netif_rx()) our stack,
340  *      so make some cleanups, and perform accounting.
341  *      Note: this accounting is not SMP safe.
342  */
343 static inline void skb_tunnel_rx(struct sk_buff *skb, struct net_device *dev,
344                                  struct net *net)
345 {
346         /* TODO : stats should be SMP safe */
347         dev->stats.rx_packets++;
348         dev->stats.rx_bytes += skb->len;
349         __skb_tunnel_rx(skb, dev, net);
350 }
351
352 int dst_discard_sk(struct sock *sk, struct sk_buff *skb);
353 static inline int dst_discard(struct sk_buff *skb)
354 {
355         return dst_discard_sk(skb->sk, skb);
356 }
357 void *dst_alloc(struct dst_ops *ops, struct net_device *dev, int initial_ref,
358                 int initial_obsolete, unsigned short flags);
359 void __dst_free(struct dst_entry *dst);
360 struct dst_entry *dst_destroy(struct dst_entry *dst);
361
362 static inline void dst_free(struct dst_entry *dst)
363 {
364         if (dst->obsolete > 0)
365                 return;
366         if (!atomic_read(&dst->__refcnt)) {
367                 dst = dst_destroy(dst);
368                 if (!dst)
369                         return;
370         }
371         __dst_free(dst);
372 }
373
374 static inline void dst_rcu_free(struct rcu_head *head)
375 {
376         struct dst_entry *dst = container_of(head, struct dst_entry, rcu_head);
377         dst_free(dst);
378 }
379
380 static inline void dst_confirm(struct dst_entry *dst)
381 {
382         dst->pending_confirm = 1;
383 }
384
385 static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,
386                                    struct sk_buff *skb)
387 {
388         const struct hh_cache *hh;
389
390         if (dst->pending_confirm) {
391                 unsigned long now = jiffies;
392
393                 dst->pending_confirm = 0;
394                 /* avoid dirtying neighbour */
395                 if (n->confirmed != now)
396                         n->confirmed = now;
397         }
398
399         hh = &n->hh;
```

```
400         if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)
401                 return neigh_hh_output(hh, skb);
402         else
403                 return n->output(n, skb);
404 }
405
406 static inline struct neighbour *dst_neigh_lookup(const struct dst_entry *dst, const void *daddr)
407 {
408         struct neighbour *n = dst->ops->neigh_lookup(dst, NULL, daddr);
409         return IS_ERR(n) ? NULL : n;
410 }
411
412 static inline struct neighbour *dst_neigh_lookup_skb(const struct dst_entry *dst,
413                                                      struct sk_buff *skb)
414 {
415         struct neighbour *n =  dst->ops->neigh_lookup(dst, skb, NULL);
416         return IS_ERR(n) ? NULL : n;
417 }
418
419 static inline void dst_link_failure(struct sk_buff *skb)
420 {
421         struct dst_entry *dst = skb_dst(skb);
422         if (dst && dst->ops && dst->ops->link_failure)
423                 dst->ops->link_failure(skb);
424 }
425
426 static inline void dst_set_expires(struct dst_entry *dst, int timeout)
427 {
428         unsigned long expires = jiffies + timeout;
429
430         if (expires == 0)
431                 expires = 1;
432
433         if (dst->expires == 0 || time_before(expires, dst->expires))
434                 dst->expires = expires;
435 }
436
437 /* Output packet to network from transport.  */
438 static inline int dst_output_sk(struct sock *sk, struct sk_buff *skb)
439 {
440         return skb_dst(skb)->output(sk, skb);
441 }
442 static inline int dst_output(struct sk_buff *skb)
443 {
444         return dst_output_sk(skb->sk, skb);
445 }
446
447 /* Input packet from network to transport.  */
448 static inline int dst_input(struct sk_buff *skb)
449 {
450         return skb_dst(skb)->input(skb);
451 }
452
453 static inline struct dst_entry *dst_check(struct dst_entry *dst, u32 cookie)
454 {
455         if (dst->obsolete)
456                 dst = dst->ops->check(dst, cookie);
457         return dst;
458 }
459
460 void dst_init(void);
461
462 /* Flags for xfrm_lookup flags argument. */
463 enum {
464         XFRM_LOOKUP_ICMP = 1 << 0,
465         XFRM_LOOKUP_QUEUE = 1 << 1,
466         XFRM_LOOKUP_KEEP_DST_REF = 1 << 2,
467 };
468
469 struct flowi;
```

```
470 #ifndef CONFIG_XFRM
471 static inline struct dst_entry *xfrm_lookup(struct net *net,
472                                             struct dst_entry *dst_orig,
473                                             const struct flowi *fl, struct sock *sk,
474                                             int flags)
475 {
476         return dst_orig;
477 }
478
479 static inline struct dst_entry *xfrm_lookup_route(struct net *net,
480                                                   struct dst_entry *dst_orig,
481                                                   const struct flowi *fl,
482                                                   struct sock *sk,
483                                                   int flags)
484 {
485         return dst_orig;
486 }
487
488 static inline struct xfrm_state *dst_xfrm(const struct dst_entry *dst)
489 {
490         return NULL;
491 }
492
493 #else
494 struct dst_entry *xfrm_lookup(struct net *net, struct dst_entry *dst_orig,
495                               const struct flowi *fl, struct sock *sk,
496                               int flags);
497
498 struct dst_entry *xfrm_lookup_route(struct net *net, struct dst_entry *dst_orig,
499                                     const struct flowi *fl, struct sock *sk,
500                                     int flags);
501
502 /* skb attached with this dst needs transformation if dst->xfrm is valid */
503 static inline struct xfrm_state *dst_xfrm(const struct dst_entry *dst)
504 {
505         return dst->xfrm;
506 }
507 #endif
508
509 #endif /* _NET_DST_H */
510
```