skbuff.h

Go to the documentation of this file.

```
00001 /*
00002  *      Definitions for the 'struct sk_buff' memory handlers.
00003  *
00004  *      Authors:
00005  *              Alan Cox, <gw4pts@gw4pts.ampr.org>
00006  *              Florian La Roche, <rzsfl@rz.uni-sb.de>
00007  *
00008  *      This program is free software; you can redistribute it and/or
00009  *      modify it under the terms of the GNU General Public License
00010  *      as published by the Free Software Foundation; either version
00011  *      2 of the License, or (at your option) any later version.
00012  */
00013
00014 #ifndef _LINUX_SKBUFF_H
00015 #define _LINUX_SKBUFF_H
00016
00017 #include <linux/config.h>
00018 #include <linux/kernel.h>
00019 #include <linux/compiler.h>
00020 #include <linux/time.h>
00021 #include <linux/cache.h>
00022
00023 #include <asm/atomic.h>
00024 #include <asm/types.h>
00025 #include <linux/spinlock.h>
00026 #include <linux/mm.h>
00027 #include <linux/highmem.h>
00028 #include <linux/poll.h>
00029 #include <linux/net.h>
00030
00031 #define HAVE_ALLOC_SKB          /* For the drivers to know */
00032 #define HAVE_ALIGNABLE_SKB      /* Ditto 8)                */
00033 #define SLAB_SKB                /* Slabified skbuffs       */
00034
00035 #define CHECKSUM_NONE 0
00036 #define CHECKSUM_HW 1
00037 #define CHECKSUM_UNNECESSARY 2
00038
00039 #define SKB_DATA_ALIGN(X)       (((X) + (SMP_CACHE_BYTES - 1)) & \
00040                                  ~(SMP_CACHE_BYTES - 1))
00041 #define SKB_MAX_ORDER(X, ORDER) (((PAGE_SIZE << (ORDER)) - (X) - \
00042                                   sizeof(struct skb_shared_info)) & \
00043                                   ~(SMP_CACHE_BYTES - 1))
00044 #define SKB_MAX_HEAD(X)         (SKB_MAX_ORDER((X), 0))
00045 #define SKB_MAX_ALLOC           (SKB_MAX_ORDER(0, 2))
00046
00047 /* A. Checksumming of received packets by device.
00048  *
00049  *      NONE: device failed to checksum this packet.
00050  *              skb->csum is undefined.
00051  *
00052  *      UNNECESSARY: device parsed packet and wouldbe verified checksum.
00053  *              skb->csum is undefined.
00054  *            It is bad option, but, unfortunately, many of vendors do this.
00055  *            Apparently with secret goal to sell you new device, when you
00056  *            will add new protocol to your host. F.e. IPv6. 8)
00057  *
00058  *      HW: the most generic way. Device supplied checksum of _all_
00059  *          the packet as seen by netif_rx in skb->csum.
00060  *          NOTE: Even if device supports only some protocols, but
00061  *          is able to produce some skb->csum, it MUST use HW,
00062  *          not UNNECESSARY.
00063  *
00064  * B. Checksumming on output.
00065  *
00066  *      NONE: skb is checksummed by protocol or csum is not required.
```

```
00067  *
00068  *      HW: device is required to csum packet as seen by hard_start_xmit
00069  *      from skb->h.raw to the end and to record the checksum
00070  *      at skb->h.raw+skb->csum.
00071  *
00072  *      Device must show its capabilities in dev->features, set
00073  *      at device setup time.
00074  *      NETIF_F_HW_CSUM - it is clever device, it is able to checksum
00075  *                        everything.
00076  *      NETIF_F_NO_CSUM - loopback or reliable single hop media.
00077  *      NETIF_F_IP_CSUM - device is dumb. It is able to csum only
00078  *                        TCP/UDP over IPv4. Sigh. Vendors like this
00079  *                        way by an unknown reason. Though, see comment above
00080  *                        about CHECKSUM_UNNECESSARY. 8)
00081  *
00082  *      Any questions? No questions, good.              --ANK
00083  */
00084
00085 #ifdef __i386__
00086 #define NET_CALLER(arg) (*(((void **)&arg) - 1))
00087 #else
00088 #define NET_CALLER(arg) __builtin_return_address(0)
00089 #endif
00090
00091 #ifdef CONFIG_NETFILTER
00092 struct nf_conntrack {
00093         atomic_t use;
00094         void (*destroy)(struct nf_conntrack *);
00095 };
00096
00097 struct nf_ct_info {
00098         struct nf_conntrack *master;
00099 };
00100
00101 #ifdef CONFIG_BRIDGE_NETFILTER
00102 struct nf_bridge_info {
00103         atomic_t use;
00104         struct net_device *physindev;
00105         struct net_device *physoutdev;
00106 #if defined(CONFIG_VLAN_8021Q) || defined(CONFIG_VLAN_8021Q_MODULE)
00107         struct net_device *netoutdev;
00108 #endif
00109         unsigned int mask;
00110         unsigned long data[32 / sizeof(unsigned long)];
00111 };
00112 #endif
00113
00114 #endif
00115
00116 struct sk_buff_head {
00117         /* These two members must be first. */
00118         struct sk_buff  *next;
00119         struct sk_buff  *prev;
00120
00121         __u32           qlen;
00122         spinlock_t      lock;
00123 };
00124
00125 struct sk_buff;
00126
00127 /* To allow 64K frame to be packed as single skb without frag_list */
00128 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
00129
00130 typedef struct skb_frag_struct skb_frag_t;
00131
00132 struct skb_frag_struct {
00133         struct page *page;
00134         __u16 page_offset;
00135         __u16 size;
00136 };
00137
00138 /* This data is invariant across clones and lives at
00139  * the end of the header data, ie. at skb->end.
00140  */
00141 struct skb_shared_info {
00142         atomic_t        dataref;
```

```
00143         unsigned int    nr_frags;
00144         unsigned short  tso_size;
00145         unsigned short  tso_segs;
00146         struct sk_buff  *frag_list;
00147         skb_frag_t      frags[MAX_SKB_FRAGS];
00148 };
00149
00191 struct sk_buff {
00192         /* These two members must be first. */
00193         struct sk_buff          *next;
00194         struct sk_buff          *prev;
00195
00196         struct sk_buff_head     *list;
00197         struct sock             *sk;
00198         struct timeval          stamp;
00199         struct net_device       *dev;
00200         struct net_device       *real_dev;
00201
00202         union {
00203                 struct tcphdr   *th;
00204                 struct udphdr   *uh;
00205                 struct icmphdr  *icmph;
00206                 struct igmphdr  *igmph;
00207                 struct iphdr    *ipiph;
00208                 struct ipv6hdr  *ipv6h;
00209                 unsigned char   *raw;
00210         } h;
00211
00212         union {
00213                 struct iphdr    *iph;
00214                 struct ipv6hdr  *ipv6h;
00215                 struct arphdr   *arph;
00216                 unsigned char   *raw;
00217         } nh;
00218
00219         union {
00220                 struct ethhdr   *ethernet;
00221                 unsigned char   *raw;
00222         } mac;
00223
00224         struct  dst_entry       *dst;
00225         struct  sec_path        *sp;
00226
00227         /*
00228          * This is the control buffer. It is free to use for every
00229          * layer. Please put your private variables there. If you
00230          * want to keep them across layers you have to do a skb_clone()
00231          * first. This is owned by whoever has the skb queued ATM.
00232          */
00233         char                    cb[48];
00234
00235         unsigned int            len,
00236                                 data_len,
00237                                 mac_len,
00238                                 csum;
00239         unsigned char           local_df,
00240                                 cloned,
00241                                 pkt_type,
00242                                 ip_summed;
00243         __u32                   priority;
00244         unsigned short          protocol,
00245                                 security;
00246
00247         void                    (*destructor)(struct sk_buff *skb);
00248 #ifdef CONFIG_NETFILTER
00249         unsigned long           nfmark;
00250         __u32                   nfcache;
00251         struct nf_ct_info       *nfct;
00252 #ifdef CONFIG_NETFILTER_DEBUG
00253         unsigned int            nf_debug;
00254 #endif
00255 #ifdef CONFIG_BRIDGE_NETFILTER
00256         struct nf_bridge_info   *nf_bridge;
00257 #endif
00258 #endif /* CONFIG_NETFILTER */
00259 #if defined(CONFIG_HIPPI)
```

```
00260         union {
00261                 __u32           ifield;
00262         } private;
00263 #endif
00264 #ifdef CONFIG_NET_SCHED
00265         __u32                   tc_index;               /* traffic control index */
00266 #endif
00267
00268         /* These elements must be at the end, see alloc_skb() for details.  */
00269         unsigned int            truesize;
00270         atomic_t                users;
00271         unsigned char           *head,
00272                                 *data,
00273                                 *tail,
00274                                 *end;
00275 };
00276
00277 #ifdef __KERNEL__
00278 /*
00279  *      Handling routines are only of interest to the kernel
00280  */
00281 #include <linux/slab.h>
00282
00283 #include <asm/system.h>
00284
00285 extern void             __kfree_skb(struct sk_buff *skb);
00286 extern struct sk_buff *alloc_skb(unsigned int size, int priority);
00287 extern void             kfree_skbmem(struct sk_buff *skb);
00288 extern struct sk_buff *skb_clone(struct sk_buff *skb, int priority);
00289 extern struct sk_buff *skb_copy(const struct sk_buff *skb, int priority);
00290 extern struct sk_buff *pskb_copy(struct sk_buff *skb, int gfp_mask);
00291 extern int              pskb_expand_head(struct sk_buff *skb,
00292                                          int nhead, int ntail, int gfp_mask);
00293 extern struct sk_buff *skb_realloc_headroom(struct sk_buff *skb,
00294                                             unsigned int headroom);
00295 extern struct sk_buff *skb_copy_expand(const struct sk_buff *skb,
00296                                        int newheadroom, int newtailroom,
00297                                        int priority);
00298 extern struct sk_buff *         skb_pad(struct sk_buff *skb, int pad);
00299 #define dev_kfree_skb(a)        kfree_skb(a)
00300 extern void           skb_over_panic(struct sk_buff *skb, int len,
00301                                      void *here);
00302 extern void           skb_under_panic(struct sk_buff *skb, int len,
00303                                       void *here);
00304
00305 /* Internal */
00306 #define skb_shinfo(SKB)         ((struct skb_shared_info *)((SKB)->end))
00307
00314 static inline int skb_queue_empty(const struct sk_buff_head *list)
00315 {
00316         return list->next == (struct sk_buff *)list;
00317 }
00318
00326 static inline struct sk_buff *skb_get(struct sk_buff *skb)
00327 {
00328         atomic_inc(&skb->users);
00329         return skb;
00330 }
00331
00332 /*
00333  * If users == 1, we are the only owner and are can avoid redundant
00334  * atomic change.
00335  */
00336
00344 static inline void kfree_skb(struct sk_buff *skb)
00345 {
00346         if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
00347                 __kfree_skb(skb);
00348 }
00349
00350 /* Use this if you didn't touch the skb state [for fast switching] */
00351 static inline void kfree_skb_fast(struct sk_buff *skb)
00352 {
00353         if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
00354                 kfree_skbmem(skb);
00355 }
```

```
00356
00365 static inline int skb_cloned(const struct sk_buff *skb)
00366 {
00367         return skb->cloned && atomic_read(&skb_shinfo(skb)->dataref) != 1;
00368 }
00369
00377 static inline int skb_shared(const struct sk_buff *skb)
00378 {
00379         return atomic_read(&skb->users) != 1;
00380 }
00381
00395 static inline struct sk_buff *skb_share_check(struct sk_buff *skb, int pri)
00396 {
00397         might_sleep_if(pri & __GFP_WAIT);
00398         if (skb_shared(skb)) {
00399                 struct sk_buff *nskb = skb_clone(skb, pri);
00400                 kfree_skb(skb);
00401                 skb = nskb;
00402         }
00403         return skb;
00404 }
00405
00406 /*
00407  *      Copy shared buffers into a new sk_buff. We effectively do COW on
00408  *      packets to handle cases where we have a local reader and forward
00409  *      and a couple of other messy ones. The normal one is tcpdumping
00410  *      a packet thats being forwarded.
00411  */
00412
00426 static inline struct sk_buff *skb_unshare(struct sk_buff *skb, int pri)
00427 {
00428         might_sleep_if(pri & __GFP_WAIT);
00429         if (skb_cloned(skb)) {
00430                 struct sk_buff *nskb = skb_copy(skb, pri);
00431                 kfree_skb(skb); /* Free our shared copy */
00432                 skb = nskb;
00433         }
00434         return skb;
00435 }
00436
00450 static inline struct sk_buff *skb_peek(struct sk_buff_head *list_)
00451 {
00452         struct sk_buff *list = ((struct sk_buff *)list_)->next;
00453         if (list == (struct sk_buff *)list_)
00454                 list = NULL;
00455         return list;
00456 }
00457
00471 static inline struct sk_buff *skb_peek_tail(struct sk_buff_head *list_)
00472 {
00473         struct sk_buff *list = ((struct sk_buff *)list_)->prev;
00474         if (list == (struct sk_buff *)list_)
00475                 list = NULL;
00476         return list;
00477 }
00478
00485 static inline __u32 skb_queue_len(const struct sk_buff_head *list_)
00486 {
00487         return list_->qlen;
00488 }
00489
00490 static inline void skb_queue_head_init(struct sk_buff_head *list)
00491 {
00492         spin_lock_init(&list->lock);
00493         list->prev = list->next = (struct sk_buff *)list;
00494         list->qlen = 0;
00495 }
00496
00497 /*
00498  *      Insert an sk_buff at the start of a list.
00499  *
00500  *      The "__skb_xxxx()" functions are the non-atomic ones that
00501  *      can only be called with interrupts disabled.
00502  */
00503
00514 extern void skb_queue_head(struct sk_buff_head *list, struct sk_buff *newsk);
```

```
00515 static inline void __skb_queue_head(struct sk_buff_head *list,
00516                                       struct sk_buff *newsk)
00517 {
00518         struct sk_buff *prev, *next;
00519
00520         newsk->list = list;
00521         list->qlen++;
00522         prev = (struct sk_buff *)list;
00523         next = prev->next;
00524         newsk->next = next;
00525         newsk->prev = prev;
00526         next->prev  = prev->next = newsk;
00527 }
00528
00539 extern void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk);
00540 static inline void __skb_queue_tail(struct sk_buff_head *list,
00541                                     struct sk_buff *newsk)
00542 {
00543         struct sk_buff *prev, *next;
00544
00545         newsk->list = list;
00546         list->qlen++;
00547         next = (struct sk_buff *)list;
00548         prev = next->prev;
00549         newsk->next = next;
00550         newsk->prev = prev;
00551         next->prev  = prev->next = newsk;
00552 }
00553
00554
00563 extern struct sk_buff *skb_dequeue(struct sk_buff_head *list);
00564 static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
00565 {
00566         struct sk_buff *next, *prev, *result;
00567
00568         prev = (struct sk_buff *) list;
00569         next = prev->next;
00570         result = NULL;
00571         if (next != prev) {
00572                 result       = next;
00573                 next         = next->next;
00574                 list->qlen--;
00575                 next->prev   = prev;
00576                 prev->next   = next;
00577                 result->next = result->prev = NULL;
00578                 result->list = NULL;
00579         }
00580         return result;
00581 }
00582
00583
00584 /*
00585  *      Insert a packet on a list.
00586  */
00587 extern void        skb_insert(struct sk_buff *old, struct sk_buff *newsk);
00588 static inline void __skb_insert(struct sk_buff *newsk,
00589                                 struct sk_buff *prev, struct sk_buff *next,
00590                                 struct sk_buff_head *list)
00591 {
00592         newsk->next = next;
00593         newsk->prev = prev;
00594         next->prev  = prev->next = newsk;
00595         newsk->list = list;
00596         list->qlen++;
00597 }
00598
00599 /*
00600  *      Place a packet after a given packet in a list.
00601  */
00602 extern void        skb_append(struct sk_buff *old, struct sk_buff *newsk);
00603 static inline void __skb_append(struct sk_buff *old, struct sk_buff *newsk)
00604 {
00605         __skb_insert(newsk, old, old->next, old->list);
00606 }
00607
00608 /*
```

```
00609  * remove sk_buff from list. _Must_ be called atomically, and with
00610  * the list known..
00611  */
00612 extern void        skb_unlink(struct sk_buff *skb);
00613 static inline void __skb_unlink(struct sk_buff *skb, struct sk_buff_head *list)
00614 {
00615         struct sk_buff *next, *prev;
00616
00617         list->qlen--;
00618         next       = skb->next;
00619         prev       = skb->prev;
00620         skb->next  = skb->prev = NULL;
00621         skb->list  = NULL;
00622         next->prev = prev;
00623         prev->next = next;
00624 }
00625
00626
00627 /* XXX: more streamlined implementation */
00628
00637 extern struct sk_buff *skb_dequeue_tail(struct sk_buff_head *list);
00638 static inline struct sk_buff *__skb_dequeue_tail(struct sk_buff_head *list)
00639 {
00640         struct sk_buff *skb = skb_peek_tail(list);
00641         if (skb)
00642                 __skb_unlink(skb, list);
00643         return skb;
00644 }
00645
00646
00647 static inline int skb_is_nonlinear(const struct sk_buff *skb)
00648 {
00649         return skb->data_len;
00650 }
00651
00652 static inline unsigned int skb_headlen(const struct sk_buff *skb)
00653 {
00654         return skb->len - skb->data_len;
00655 }
00656
00657 static inline int skb_pagelen(const struct sk_buff *skb)
00658 {
00659         int i, len = 0;
00660
00661         for (i = (int)skb_shinfo(skb)->nr_frags - 1; i >= 0; i--)
00662                 len += skb_shinfo(skb)->frags[i].size;
00663         return len + skb_headlen(skb);
00664 }
00665
00666 static inline void skb_fill_page_desc(struct sk_buff *skb, int i, struct page *page, int off, int size)
00667 {
00668         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
00669         frag->page = page;
00670         frag->page_offset = off;
00671         frag->size = size;
00672         skb_shinfo(skb)->nr_frags = i+1;
00673 }
00674
00675 #define SKB_PAGE_ASSERT(skb)    BUG_ON(skb_shinfo(skb)->nr_frags)
00676 #define SKB_FRAG_ASSERT(skb)    BUG_ON(skb_shinfo(skb)->frag_list)
00677 #define SKB_LINEAR_ASSERT(skb)  BUG_ON(skb_is_nonlinear(skb))
00678
00679 /*
00680  *      Add data to an sk_buff
00681  */
00682 static inline unsigned char *__skb_put(struct sk_buff *skb, unsigned int len)
00683 {
00684         unsigned char *tmp = skb->tail;
00685         SKB_LINEAR_ASSERT(skb);
00686         skb->tail += len;
00687         skb->len  += len;
00688         return tmp;
00689 }
00690
00700 static inline unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
00701 {
```

```
00702         unsigned char *tmp = skb->tail;
00703         SKB_LINEAR_ASSERT(skb);
00704         skb->tail += len;
00705         skb->len  += len;
00706         if (unlikely(skb->tail>skb->end))
00707                 skb_over_panic(skb, len, current_text_addr());
00708         return tmp;
00709 }
00710
00711 static inline unsigned char *__skb_push(struct sk_buff *skb, unsigned int len)
00712 {
00713         skb->data -= len;
00714         skb->len  += len;
00715         return skb->data;
00716 }
00717
00727 static inline unsigned char *skb_push(struct sk_buff *skb, unsigned int len)
00728 {
00729         skb->data -= len;
00730         skb->len  += len;
00731         if (unlikely(skb->data<skb->head))
00732                 skb_under_panic(skb, len, current_text_addr());
00733         return skb->data;
00734 }
00735
00736 static inline unsigned char *__skb_pull(struct sk_buff *skb, unsigned int len)
00737 {
00738         skb->len -= len;
00739         BUG_ON(skb->len < skb->data_len);
00740         return skb->data += len;
00741 }
00742
00753 static inline unsigned char *skb_pull(struct sk_buff *skb, unsigned int len)
00754 {
00755         return unlikely(len > skb->len) ? NULL : __skb_pull(skb, len);
00756 }
00757
00758 extern unsigned char *__pskb_pull_tail(struct sk_buff *skb, int delta);
00759
00760 static inline unsigned char *__pskb_pull(struct sk_buff *skb, unsigned int len)
00761 {
00762         if (len > skb_headlen(skb) &&
00763             !__pskb_pull_tail(skb, len-skb_headlen(skb)))
00764                 return NULL;
00765         skb->len -= len;
00766         return skb->data += len;
00767 }
00768
00769 static inline unsigned char *pskb_pull(struct sk_buff *skb, unsigned int len)
00770 {
00771         return unlikely(len > skb->len) ? NULL : __pskb_pull(skb, len);
00772 }
00773
00774 static inline int pskb_may_pull(struct sk_buff *skb, unsigned int len)
00775 {
00776         if (likely(len <= skb_headlen(skb)))
00777                 return 1;
00778         if (unlikely(len > skb->len))
00779                 return 0;
00780         return __pskb_pull_tail(skb, len-skb_headlen(skb)) != NULL;
00781 }
00782
00789 static inline int skb_headroom(const struct sk_buff *skb)
00790 {
00791         return skb->data - skb->head;
00792 }
00793
00800 static inline int skb_tailroom(const struct sk_buff *skb)
00801 {
00802         return skb_is_nonlinear(skb) ? 0 : skb->end - skb->tail;
00803 }
00804
00813 static inline void skb_reserve(struct sk_buff *skb, unsigned int len)
00814 {
00815         skb->data += len;
00816         skb->tail += len;
```

```
00817 }
00818
00819 extern int ___pskb_trim(struct sk_buff *skb, unsigned int len, int realloc);
00820
00821 static inline void __skb_trim(struct sk_buff *skb, unsigned int len)
00822 {
00823         if (!skb->data_len) {
00824                 skb->len  = len;
00825                 skb->tail = skb->data + len;
00826         } else
00827                 ___pskb_trim(skb, len, 0);
00828 }
00829
00838 static inline void skb_trim(struct sk_buff *skb, unsigned int len)
00839 {
00840         if (skb->len > len)
00841                 __skb_trim(skb, len);
00842 }
00843
00844
00845 static inline int __pskb_trim(struct sk_buff *skb, unsigned int len)
00846 {
00847         if (!skb->data_len) {
00848                 skb->len  = len;
00849                 skb->tail = skb->data+len;
00850                 return 0;
00851         }
00852         return ___pskb_trim(skb, len, 1);
00853 }
00854
00855 static inline int pskb_trim(struct sk_buff *skb, unsigned int len)
00856 {
00857         return (len < skb->len) ? __pskb_trim(skb, len) : 0;
00858 }
00859
00868 static inline void skb_orphan(struct sk_buff *skb)
00869 {
00870         if (skb->destructor)
00871                 skb->destructor(skb);
00872         skb->destructor = NULL;
00873         skb->sk         = NULL;
00874 }
00875
00884 extern void skb_queue_purge(struct sk_buff_head *list);
00885 static inline void __skb_queue_purge(struct sk_buff_head *list)
00886 {
00887         struct sk_buff *skb;
00888         while ((skb = __skb_dequeue(list)) != NULL)
00889                 kfree_skb(skb);
00890 }
00891
00904 static inline struct sk_buff *__dev_alloc_skb(unsigned int length,
00905                                               int gfp_mask)
00906 {
00907         struct sk_buff *skb = alloc_skb(length + 16, gfp_mask);
00908         if (likely(skb))
00909                 skb_reserve(skb, 16);
00910         return skb;
00911 }
00912
00925 static inline struct sk_buff *dev_alloc_skb(unsigned int length)
00926 {
00927         return __dev_alloc_skb(length, GFP_ATOMIC);
00928 }
00929
00942 static inline int skb_cow(struct sk_buff *skb, unsigned int headroom)
00943 {
00944         int delta = (headroom > 16 ? headroom : 16) - skb_headroom(skb);
00945
00946         if (delta < 0)
00947                 delta = 0;
00948
00949         if (delta || skb_cloned(skb))
00950                 return pskb_expand_head(skb, (delta + 15) & ~15, 0, GFP_ATOMIC);
00951         return 0;
00952 }
```

```
00953
00966 static inline struct sk_buff *skb_padto(struct sk_buff *skb, unsigned int len)
00967 {
00968         unsigned int size = skb->len;
00969         if (likely(size >= len))
00970                 return skb;
00971         return skb_pad(skb, len-size);
00972 }
00973
00982 extern int __skb_linearize(struct sk_buff *skb, int gfp);
00983 static inline int skb_linearize(struct sk_buff *skb, int gfp)
00984 {
00985         return __skb_linearize(skb, gfp);
00986 }
00987
00988 static inline void *kmap_skb_frag(const skb_frag_t *frag)
00989 {
00990 #ifdef CONFIG_HIGHMEM
00991         BUG_ON(in_irq());
00992
00993         local_bh_disable();
00994 #endif
00995         return kmap_atomic(frag->page, KM_SKB_DATA_SOFTIRQ);
00996 }
00997
00998 static inline void kunmap_skb_frag(void *vaddr)
00999 {
01000         kunmap_atomic(vaddr, KM_SKB_DATA_SOFTIRQ);
01001 #ifdef CONFIG_HIGHMEM
01002         local_bh_enable();
01003 #endif
01004 }
01005
01006 #define skb_queue_walk(queue, skb) \
01007                 for (skb = (queue)->next, prefetch(skb->next);  \
01008                      (skb != (struct sk_buff *)(queue));        \
01009                      skb = skb->next, prefetch(skb->next))
01010
01011
01012 extern struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags,
01013                                          int noblock, int *err);
01014 extern unsigned int    datagram_poll(struct file *file, struct socket *sock,
01015                                      struct poll_table_struct *wait);
01016 extern int             skb_copy_datagram(const struct sk_buff *from,
01017                                          int offset, char *to, int size);
01018 extern int             skb_copy_datagram_iovec(const struct sk_buff *from,
01019                                                int offset, struct iovec *to,
01020                                                int size);
01021 extern int             skb_copy_and_csum_datagram(const struct sk_buff *skb,
01022                                                   int offset, u8 *to, int len,
01023                                                   unsigned int *csump);
01024 extern int             skb_copy_and_csum_datagram_iovec(const
01025                                                         struct sk_buff *skb,
01026                                                         int hlen,
01027                                                         struct iovec *iov);
01028 extern void            skb_free_datagram(struct sock *sk, struct sk_buff *skb);
01029 extern unsigned int    skb_checksum(const struct sk_buff *skb, int offset,
01030                                     int len, unsigned int csum);
01031 extern int             skb_copy_bits(const struct sk_buff *skb, int offset,
01032                                      void *to, int len);
01033 extern unsigned int    skb_copy_and_csum_bits(const struct sk_buff *skb,
01034                                               int offset, u8 *to, int len,
01035                                               unsigned int csum);
01036 extern void            skb_copy_and_csum_dev(const struct sk_buff *skb, u8 *to);
01037
01038 extern void skb_init(void);
01039 extern void skb_add_mtu(int mtu);
01040
01041 struct tux_req_struct;
01042
01043 #ifdef CONFIG_NETFILTER
01044 static inline void nf_conntrack_put(struct nf_ct_info *nfct)
01045 {
01046         if (nfct && atomic_dec_and_test(&nfct->master->use))
01047                 nfct->master->destroy(nfct->master);
01048 }
```

```
01049 static inline void nf_conntrack_get(struct nf_ct_info *nfct)
01050 {
01051         if (nfct)
01052                 atomic_inc(&nfct->master->use);
01053 }
01054
01055 #ifdef CONFIG_BRIDGE_NETFILTER
01056 static inline void nf_bridge_put(struct nf_bridge_info *nf_bridge)
01057 {
01058         if (nf_bridge && atomic_dec_and_test(&nf_bridge->use))
01059                 kfree(nf_bridge);
01060 }
01061 static inline void nf_bridge_get(struct nf_bridge_info *nf_bridge)
01062 {
01063         if (nf_bridge)
01064                 atomic_inc(&nf_bridge->use);
01065 }
01066 #endif
01067
01068 #endif
01069
01070 #endif  /* __KERNEL__ */
01071 #endif  /* _LINUX_SKBUFF_H */
```
