上层传给内核的tcp数据包如何能够快速的发送下去。

Struct inet\_connection\_sock {

U32 icsk\_pmtu\_cookie;

Struct {

Int search\_high;

Int search\_low;

Int enabled;

Int probe\_size;

U32 probe\_timestamp;

}icsk\_mtup;

};

Struct tcp\_sock {

U32 mss\_cache;

U16 tcp\_header\_len;

U16 gso\_segs;

};

Struct page\_frag {

Struct page \*page;

U16 offset;

U16 size;

};

Struct dst\_ops {

Unsigned int (\*mtu)(const struct dst\_entry \*);

};

Struct dst\_entry {

Struct net\_device \*dev;

Struct dst\_ops \*ops;

Unsigned long \_metrics;

};

Struct rtable {

U32 rt\_pmtu:31;

};

Struct sock {

refcount\_t sk\_wmem\_alloc;/\*代表离开tcp层，在网卡发送出去之前的总报文数目\*/

unsigned char \*head;

unsigned char \*end;/\*仅是32bit系统是unsiged char \*定义,struct skb\_shared\_info即在skb->end起始的位置\*/

unsigned char \*tail;/\*仅是32bit系统是unsigned char\*定义\*/

unsigned char \*data;

u8 cloned:1;

u8 nohdr:1;

u8 head\_frag:1;

struct page\_frag sk\_frag;

gfp\_t sk\_allocation;

struct dst\_entry \_\_rcu \*sk\_dst\_cache;

unsigned int sk\_gso\_max\_size;

u16 sk\_gso\_max\_segs;

struct sk\_buff \*sk\_tx\_skb\_cache;

};

Struct sk\_buff\_fclones {

Struct sk\_buff skb1;

Struct sk\_buff skb2;

Refcount\_t fclone\_ref;

};

Skb\_frag\_t {

Struct {

Struct page \*p;

}page;

U16 page\_offset;

U16 size;

};

Struct skb\_shared\_info {

Atomic\_t dataref;

U8 nr\_frags;

Skb\_frag\_t frags[16];

Struct sk\_buff \*frag\_list;

Void \*destructor\_arg;

};

Struct sk\_buff {

U8 fclone:2;

Char cb[48] \_\_aligned(8)；

Union {

Struct {

Struct sk\_buff \*next;

Struct sk\_buff \*prev;

}；

Struct rb\_node rbnode;

Struct list\_head list;

};

};

Struct sk\_buff\_head {

Struct sk\_buff \*next;

Struct sk\_buff \*prev;

U32 qlen;

}

**Sk\_wmem\_alloc**

1. Tcp\_transmit\_skb中，会调用如下函数:
2. skb->destrutor = skb\_is\_tcp\_pure\_ack(skb) ? \_\_sock\_wfree:tcp\_wfree;

初始化skb的destrutor函数，在destrutor函数中处理skb的释放，其中会减减sk\_wmem\_alloc。

1. refcount\_add(skb->truesize,&sk->sk\_wmem\_alloc);

增加wk\_wmem\_alloc的值。

1. 在网卡的发包完成处理函数中，一般是netdev\_napi\_poll的回调函数，会调用dev\_kfree\_skb\_any(skb)释放skb，考虑一个问题，dev\_kfree\_skb\_any会真正释放skb吗，tcp只有在收到对应的包的ack后才应该释放对应的资源吧。探究如下:

dev\_kfree\_skb\_any->

dev\_kfree\_skb->

consume\_skb->

1. Skb\_unref(判断skb->users成员是否是1，如果是1，返回true，说明可释放skb，否则，返回false，说明不可释放skb)->
2. \_\_kfree\_skb
3. Skb\_release\_all(skb)
4. Skb\_release\_head\_state(skb)
5. Skb\_dst\_drop
6. 如果skb的destrutor存在，则调用skb->desctrutor即tcp\_wmem处理
7. Skb\_ext\_put，需要skb的active\_extensions置位才继续处理。
8. 如果skb->head不为空，进入skb\_release\_data流程:
9. 如果skb->cloned置位且struct skb\_shared\_info的dataref减去一个值后(减去值的大小根据skb->nohdr是否置位体现)不为零，则直接返回
10. 根据struct skb\_shared\_info的nr\_frags遍历struct skb\_shared\_info的frags，然后调用\_\_skb\_frag\_unref释放struct skb\_shared\_info的frags
11. 释放struct skb\_shared\_info的frag\_list
12. 调用skb\_zcopy\_clear
13. 调用skb\_free\_head
14. Kfree\_skbmem

从上述释放skb的过程看，无论如何都是要释放skb的数据的，所以是否在发送之前clone了skb呢，跟踪tcp\_transmit\_skb，果然，在开始的位置，判断是否clone\_it，如果是，则会将传入的skb替换成clone出来的skb。

**tcp\_sendmsg**

tcp\_sendmsg有zerocopy和非zerocopy两种方式，我们先考虑非zerocopy的。

1. tcp\_send\_mss，获取size\_goal和mss\_now,cp\_send\_mss的调用流程如下
2. tcp\_current\_mss(sk)
3. 获取struct tcp\_sock的mss\_cache成员作为mss\_now
4. 如果struct sock的dst不为空，则调用dst->ops->mtu，即ipv4\_mtu获取

Mtu，ipv4\_mtu的主要流程如下：

1. 判断struct rtable的rt\_pmtu是否不为零，不为零，则返回
2. 判断struct dst\_entry的\_metrics数组的RTAX\_MTU成员是否不为零，不为零，则返回
3. 获取Struct dst\_entry的struct net\_device成员 的mtu，在此判断ip\_mtu\_locked和rt->rt\_uses\_gateway和mtu是否大于576，如果是的，则mtu等于576
4. 返回mtu

如果通过ipv4\_mtu获取的mtu和struct inet\_connection\_sock中缓存的icsk\_pmtu\_cookie不一样，则调用tcp\_sync\_mss替换mss\_now，tcp\_sync\_mss调用流程如下:

1. Tcp\_sync\_mss传入参数为struct sock和通过ipv4\_mtu获取的mtu
2. 判断icsk->icsk\_mtup.search\_high是否大于传入的mtu，如果是，则赋值传入的mtu到icsk->icsk\_mtup.search\_high
3. 调用tcp\_mtu\_to\_mss(sk,pmtu)，计算mss\_now
4. 调用tcp\_bound\_to\_half\_wnd，计算mss\_now
5. 存储传入的mtu到icsk->icsk\_pmtu\_cookie中
6. 存储新的mss\_now到struct tcp\_sock的mss\_now中
7. 返回mss\_now
8. 通过tcp\_established\_options计算header\_len，再次通过得到的header\_len计算mss\_now
9. 返回mss\_now
10. 通过调用tcp\_xmit\_size\_goal计算size\_goal
11. 如果当前是OOB传输，则返回mss\_now作为size\_goal
12. New\_size\_goal等于sk->sk\_gso\_max\_size-1-MAX\_TCP\_HEADER，sk->sk\_gso\_max\_size是在tcp连接时通过sk\_setup\_caps设置的。
13. New\_size\_goal 等于tcp\_bound\_to\_half\_wnd(tp,size\_goal)
14. Size\_goal 等于struct tcp\_sock的gso\_segs成员乘以mss\_now
15. 判断new\_size\_goal是否小于size\_goal或者new\_size\_goal是否大于size\_goal+mss\_now，如果是，则重新初始化struct tcp\_sock的gso\_segs
16. 返回size\_goal
17. 进入msg\_data\_left(msg)循环中，有两个变量，copy和copied
18. Copy赋值为0
19. 首先获取struct sock的sk\_write\_queue的tail，判断是否为空
20. 如果不为空，则copy赋值为size\_goal-skb->len
21. 如果copy小于等于0，则分配调用sk\_stream\_alloc\_skb分配skb，sk\_stream\_alloc\_skb的传入参数分别是sk，0， sk->sk\_allocation(inet\_create->sock\_init\_data->sk->sk\_allocation ，sk->sk\_allocation初始化为GFP\_KERNEL)，first\_skb（如果sk的tcp\_rtx\_queue和sk的sk\_write\_qeueu都为空，则first\_skb为1），sk\_tream\_alloc\_skb的调用过程如下（不考虑内存不够的情况）:
22. 如果传入的size为0，则判断sk->sk\_tx\_skb\_cache是否为空，如果不为空，则直接返回sk->sk\_tx\_skb\_cache，如果为空，则进入下一步
23. 传入大小4字节对齐
24. 调用alloc\_skb\_fclone分配struct sk\_buff，传入size+sk->sk\_prot->max\_header，gfp为传入的GFP\_KERNEL，alloc\_skb\_fclone调用流程如下：
25. \_\_alloc\_skb(size,GFP\_KERNEL,SKB\_ALLOC\_FCLONE,NUMA\_NO \_NODE)
26. struct kmem\_cache根据传入的SKB\_ALLOC\_FCLONE判断为skbuff\_fclone\_cache
27. 从skbuff\_fclone\_cache中分配skb
28. 传入size cache line大小对齐并赋值给size
29. Size+=cache line大小对齐的struct skb\_shared\_info大小
30. kmalloc\_reserve分配上述size大小的data，kmalloc\_reserve传入pfmemalloc指针可以告诉调用者是不是使用的emergency pfmemalloc
31. ksize获取kmalloc\_reserve分配的实际的大小，并调用SKB\_WITH\_OVERHEAD来得到kmalloc\_reserve的实际分配的大小减去struct skb\_shared\_info cache line对齐的大小赋值给size
32. 初始化skb的truesize成员为刚刚得到的size的大小+struct sk\_buff cache line大小对齐后的大小+struct skb\_shared\_info cache line大小对齐后的大小
33. 初始化skb的pfmemalloc成员为从kmalloc\_reserve获取的pfmemalloc变量
34. 初始化skb head成员为kmalloc\_reserve分配的data指针
35. 初始化skb data成员为kmalloc\_reserve分配的data指针
36. 初始化skb tail成员为skb 的data指针
37. 初始化skb end成员为 第g部获取的size大小+skb data指针
38. 获取struct sk\_buff的struct skb\_shared\_info
39. 初始化struct skb\_shared\_info的dataref为1
40. 判断传入flag是否是SKB\_ALLOC\_FCLONE，如果是，skb的fclone成员赋值为SKB\_FCLONE\_ORIG，fclone的fclone\_ref赋值为1，fclone的skb2.fclone赋值为SKB\_FCLONE\_CLONE
41. 如果skb分配成功，则判断force\_schedule是否置位，如果传入的first\_skb为1，则force\_schedule置位，会进入如下流程
42. Mem\_scheduled 置为true
43. 调用sk\_forced\_mem\_schedule，传入大小为skb的truesize,sk\_forced\_mem\_schedule的主要处理过程如下:
    * 1. 调用sk\_mem\_pages向上取skb truesize的PAGE\_SIZE的个数
      2. Sk->sk\_forward\_alloc += skb trusize PAGE\_SIZE的个数乘以PAGE\_SIZE
      3. 增加sk->sk\_prot->memory\_allocated的大小为skb truesize的PAGE\_SIZE向上取值的个数
44. 如果mem\_scheduled置位，则会进入如下处理流程
45. 调用skb\_reserved,reserved sk->sk\_prot->max\_header大小，主要赋值如下：
    1. Skb->data+=sk->sk\_prot->max\_header
    2. Skb->tail+= sk->sk\_prot->max\_header
46. Skb->reserved\_tailroom置为skb->end-skb->tail –size大小，其实就是由于cache line对齐和kmalloc\_reserve多分配出来的内存大小
47. Skb分配完以后，skb的ip\_summed置为CHECKSUM\_PARTIAL
48. 调用skb\_entail，主要处理流程如下:

A) skb的csum成员赋值为0

B)kb的cb即struct tcp\_skb\_cb的seq和end\_seq成员赋值为struct tcp\_sock的write\_seq

C)skb的cb即struct tcp\_skb\_cb的tcp\_flags置为TCPHDR\_ACK

D) skb的cb即struct tcp\_skb\_cb的sack置为0

E)调用\_\_skb\_header\_release，主要做如下两件事:

a)skb->nohdr置为1

b)skb的struct skb\_shared\_info的dataref置为1+（1<<SKB\_DATAREF\_SHIFT）

F)调用tcp\_add\_write\_queue\_tail（）

a)将skb通过调用\_\_skb\_queue\_tail加入到sk\_write\_queue中

b) 判断sk->sk\_write\_queue.next 是否是skb，什么时候出现？

G)增加sk->sk\_wmem\_queued skb->truesize;

H)调用sk\_mem\_charge

a)判断sk->sk\_prot->memory\_allocated是否为空，为空则返回

b)sk->sk\_forward\_alloc减去truesize

I)判断struct tcp\_sock的nonagle的TCP\_NAGLE\_PUSH是否置位，如果置

则清楚TCP\_NAGLE\_PUSH

J) tcp\_slow\_start\_after\_idle\_check

7) copy 赋值为size\_goal

8) 通过skb\_availroom看skb是否有剩余大小，如果有且不是零拷贝，则将用户传入的buf拷贝到skb中，显然在sk\_stream\_alloc\_skb中没有分配数据的空间，所以会走到下一个流程,拷贝到struct page\_frag中，在struct sock中有个成员变量struct page\_frag sk\_frag，主要是把用户数据拷贝到struct page\_frag sk\_frag中，流程如下：

A) 调用sk\_page\_frag\_refill，传入参数为struct sock和struct sock的sk\_frag指帧，sk\_page\_frag\_refill主要做了如下事情:

a) 调用skb\_page\_frag\_refill，传入sz大小为32

i) 判断传入的struct page\_frag中page指针是否为空，如果不为空，且page的refcount为1，则struct page\_frag的offset置为0并返回；如果page的refcount不为1，则判断struct page\_frag的offset加上传入的sz是否小于struct page\_frag的size，如果小于，则返回true。否则，put\_page，进入下一流程。

ii)判断SKB\_FRAG\_PAGE\_ORDER是否不为零且net\_high\_order\_alloc\_disable\_key是否不为0，如果是，则调用alloc\_pages分配32KB的page，并赋值32KB给struct page\_frag的size返回，如果不是，进入下一流程

iii)调用alloc\_page分配4KB大小的page，并赋值4KB给struct page\_frag的size，返回。

b) 如果skb\_page\_frag\_refill分配成功，则返回，如果分配不成功，则进入memory\_pressusr阶段

B) 调用skb\_can\_coalesce，传入skb，struct skb\_shared\_info的nr\_frags,struct page\_frag的page和struct page\_frag的offset，主要流程如下:

i)判断skb是否是zerocopy，如果是，则返回false

ii)如果struct skb\_shared\_info的nr\_frags不为0，则获取struct skb\_shared\_info的frags[nr\_frags-1],并判断传入的page是否为struct skb\_shared\_info 的frags[nr\_frags-1]的page并且struct page\_frag的offset是否等于struct skb\_shared\_info的frags[nr\_frags-1]的page\_offset+frags[nr\_frags-1]的size，如果是，则返回true，否则返回false

9) 如果skb\_can\_coalesce返回false且struct skb\_shared\_info的nr\_frags大于sysctl\_max\_skb\_frags(17)，则调用tcp\_mark\_push,即设置struct sk\_buff的cb的tcp\_flags的TCPHDR\_PSH，并设置struct tcp\_sock的pushed\_seq为当前的write\_seq,并进入new\_segment阶段重新分配skb；如果struct skb\_shared\_info的nr\_frags小于sysctl\_max\_skb\_frags(17)，则设置merge为false并继续走下去。

10)如果skb\_can\_coalesce返回true，则设置merge为true并继续走下去

11）获取copy和struct page\_frag->size – struct page\_frag->offset的较小值

12）调用sk\_wmem\_schedule，传入参数为sk和copy主要流程如下：

A)首先判断struct sock的struct proto成员的memory\_allocated是否为空，如果为空，则直接返回true

B)判断传入的copy是否小于sk的sk\_forward\_alloc，如果是，则直接返回true

C)进入\_\_sk\_mem\_schedule函数，传入参数为struct sock、copy大小和SK\_MEM\_SEND，如果\_\_sk\_mem\_schedule返回true，则sk\_wmem\_schedule返回true，主要流程如下:

a) 增加sk->sk\_forward\_alloc 传入copy 4K 向上对齐大小

b)调用\_\_sk\_mem\_raise\_allocated()，传入参数为struct sock,传入copy大小，copy大小除以4K大小，和SK\_MEM\_SEND，\_\_sk\_mem\_raise\_allocated的流程如下:

## the rack

struct tcp\_sock {

/\*time-sorted sent but un-SACKED skbs\*/

struct list\_head tsorted\_sent\_queue;

u32 reord\_seen;/\*number of data packet reordering events\*/

};

struct sk\_buff {

struct list\_head tcp\_tsorted\_anchor;

u64 skb\_mstamp\_ns;

};

struct tcp\_options\_received {

u16 saw\_tstamp:1;/\*tstamp:tcp的timestamp\*/

u16 saw\_tstamp:1;/\*Saw timestamp on last packet\*/

};

struct tcp\_sock {

struct tcp\_rack {

u64 mstamp;/\*sent time of the skb\*/

u32 rtt\_us; /\*associated rtt\*/

u32 end\_seq;

u32 last\_delivered;

u8 reo\_wnd\_steps;/\*Allowed reordering window\*/

u8 advanced:1;/\*mstamp advanced since last lost marking\*/

}rack;

u32 rcv\_wup;

struct sk\_buff \*highest\_sack; /\*skb just after the highest skb with SACKED bit set(validity guaranteed only if sacked\_out > 0)\*/

u64 tcp\_mstamp;/\*most recent packet received/sent\*/

struct tcp\_options\_received rx\_opt;

};

/\*

Header prediction.

\*/

tcp\_rack\_detect\_loss(sk,&timeout);

{

u32 reo\_wnd;

reo\_wnd = tcp\_rack\_reo\_wnd(sk);

list\_for\_each\_entry\_safe(skb,n,&tp->tsorted\_sent\_queue,tcp\_tsorted\_anchor) {

s32 remaining;

struct tcp\_skb\_cb \*scb = TCP\_SKB\_CB(skb);

remaining = tcp\_rack\_skb\_timeout(tp,skb,reo\_wnd);

if(remaining <= 0) {

if(remaining <=0) {

tcp\_mark\_skb\_lost(sk,skb);

list\_del\_init(&skb->tcp\_tsorted\_anchor);

}else {

\*reo\_timeout = max\_t(u32,\*reo\_timeout,remaining);

}

}

}

}

/\*

#define TCPCB\_RETRANS (TCPCB\_SACKED\_RETRANS | TCPCB\_EVER\_RETRANS | TCPCB\_REPAIRED)

TCPCB\_SACKED\_RETRANS:在赋值时以TCPCB\_RETRANS的形式赋值,

tcp\_retransmit\_skb->TCP\_SKB\_CB(skb)->sacked |= TCPCB\_RETRNAS;

\*/

tcp\_mark\_skb\_lost(sk,skb)

{

struct tcp\_sock \*tp =tcp\_sk(sk);

}

tcp\_skb\_timestamp\_us(skb)

{

return div\_u64(skb->skb\_mstamp\_us,NSEC\_PER\_USEC);

}

/\*

t1:mstamp;

t2:skb发送时间:

seq1：最后的end\_seq

seq2: skb的最后end\_seq

\*/

tcp\_rack\_sent\_after(t1,t2,seq1,seq2)

{

}

#define TCP\_RACK\_LOSS\_DETECTION 0x1 /\*Use RACK to detect losses\*/

#define TCP\_RACK\_STATIC\_REO\_WND 0x2 /\*Use static RACK to reo wnd\*/

#define TCP\_RACK\_NO\_DUPTHRESH 0x4 /\*Do not use DUPACK threshold in RACK\*/

tcp\_rack\_reo\_wnd(sk)

{

if(!tp->reord\_seen) {

/\*

reordering has not been observed.

\*/

if(inet\_csk(sk)->icsk\_ca\_state>=TCP\_CA\_Recovery)

return 0;

if(tp->sacked\_out>=tp->reordering &&

!(sock\_net(sk)->ipv4.sysctl\_tcp\_recovery&TCP\_RACK\_NO\_DUPTHRESH))

return 0;

}

return min((tcp\_min\_rtt(tp)>>2)\*tp->rack.reo\_wnd\_steps,

tp->srtt\_us>>3);

}

tcp\_rack\_skb\_timeout(tp,skb,reo\_wnd)

{

/\*

tcp\_stamp\_us\_delta(t1,t0):return t1-t0;

\*/

return tp->rack.rtt\_us + reo\_wnd –

tcp\_stamp\_us\_delta(tp->tcp\_mstamp,tcp\_skb\_timestamping\_us(skb));

}

## fack

tcp\_highest\_sack\_seq(tp)

{

if(!tp-> sacked\_out)

return tp->snd\_una;

}

/\*

A rate sample measures the number of (original/retransmitted) data packets delivered “delivered” over an interval of time “interval\_us”.the tcp\_rate.c code fills in the rate sample.

\*/

struct rate\_sample {

};

/\*

FLAG\_SLOWPATH | FLAG\_UPDATE\_TS\_RECENT

\*/

tcp\_ack(sk,skb,flag)

{

u32 prior\_fack;

struct rate\_sample rs = {.prior\_delivered =0};

/\*

forward acknowlegement:当前最高的ack

\*/

prior\_fack = tcp\_is\_sack(tp)?tcp\_highest\_sack\_seq(tp):tp->snd\_una;

/\*

tcp\_packets\_in\_flight:tp->packets\_out+tp->retrans\_out-

(tp->sacked\_out+tp->lost\_out)

\*/

rs.prior\_in\_flight = tcp\_packets\_in\_flight(tp);

tcp\_replace\_ts\_recent(tp,TCP\_SKB\_CB(skb)->seq);

}

tcp\_replace\_ts\_recent(tp,TCP\_SKB\_CB(skb)->seq)

{

}

tcp\_clean\_rtx\_queue(sk,pri)

for(sk->tcp\_rtx\_queue;)

{

u32 prior\_fack;

}

## \_\_tcp\_transmit\_skb

tcp\_mstamp\_refresh(tp)

{

u64 val = tcp\_clock\_ns();

tp->tcp\_clock\_cache = val;

tp->tcp\_mstamp = div\_u64(val,NSEC\_PER\_USEC);

}

## tcp\_rcv\_established

tcp\_rcv\_established(sk,skb)

{

/\*

\*/

tcp\_mstamp\_refresh(tp);

tp->rx\_opt.saw\_tstamp =0;

}